Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática,
2010

Patrick Ferreira Marques

Arquitectura concorrente para o controlo de um veículo autónomo

Concurrent architecture for control of an autonomous driving vehicle

"Simplicity is the ultimate sophistication"

— Leonardo Da Vinci

Patrick Ferreira Marques

Arquitectura concorrente para o controlo de um veículo autónomo

Concurrent architecture for control of an autonomous driving vehicle

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores de Telemática, realizada sob a orientação científica do Prof. Dr. Artur José Carneiro Pereira e do Prof. Dr. José Luís Costa Pinto de Azevedo, professores do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Doutor António Rui de Oliveira e Silva Borges
Professor Associado da Universidade de Aveiro

vogais / examiners committee

Doutor Paulo José Cerqueira Gomes da Costa
Professor Auxiliar da Faculdade de Engenharia da Universidade do Porto

Doutor Artur José Carneiro Pereira
Professor Auxiliar da Universidade de Aveiro (orientador)

Doutor José Luís Costa Pinto de Azevedo
Professor Auxiliar da Universidade de Aveiro (co-orientador)

agradecimentos

Gostava de deixar aqui os meus agradecimentos a todas as pessoas que de alguma forma participaram e/ou influenciaram a realização deste projecto. Começando pelos meus pais e tios, que sempre me deram forças para terminar este trabalho. Obrigado a todos os professores que me deram formação, especialmente aos meus orientadores pela motivação dada e ajuda prestada ao longo do último ano bem como todo o conhecimento transmitido. Também agradeço aos meus amigos que sempre ajudaram e apoiaram nos momentos mais difíceis.

Palavras-chave Robótica, Arquitectura Concorrente, Robótica Autónoma e Móvel, Condução Autónoma, Festival Nacional de Robótica

Resumo

Os robôs autónomos e os veículos não tripulados são vertentes da robótica de forte investigação durante os últimos anos, especialmente para o desenvolvimento de veículos autónomos destinados à exploração de lugares inóspitos. Um robô autónomo é uma máquina que consegue ser independente e regida pelas suas próprias leis, um sistema que consegue sobreviver num ambiente natural sem intervenção humana. Hoje em dia são muitos os sistemas disponíveis (como por exemplo GPS e visão computorizada) que ajudam os robôs a sobreviver, sendo o grande desafio integrá-los de forma a produzir um ser mais inteligente, sólido e confiável.

O Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro tem realizado, nos últimos anos, trabalho na área da condução autónoma. Um dos objectivos desse trabalho tem sido o desenvolvimento de veículos autónomos para participação na prova de condução autónoma do Festival Nacional de Robótica, na qual participa desde 2001. O software de controlo de alto nível do veículo actual assentava numa estrutura sequencial o que torna complexa a tarefa de manutenção e integração de novas funcionalidades. Actualmente existem módulos para a descrição do mundo bem como para o planeamento de trajectórias sobre esses modelos e um módulo para percepção a partir de imagem.

O objectivo deste trabalho foi o de reestruturar todo o software de alto nível, tornando-o modular e concorrente permitindo dessa forma uma melhor manutenção, actualização e evolução. Fica assim mais fácil a substituição de módulos, a incorporação de novas funcionalidades e o trabalho de equipa. A nova estrutura assenta na existência de uma memória central e partilhada, tipo *blackboard*, onde os vários módulos recolhem os dados de que necessitam e depositam os dados produzidos.

Este novo modelo arquitectural foi implementado no veículo e testado durante a edição de 2010 do Festival Nacional de Robótica, tendo alcançado o terceiro lugar. A arquitectura apresentada incorporou vários sistemas já existentes, tendo como principais vantagens a modularidade e extensibilidade dentro de um ambiente concorrente e com informação distribuída.

Keywords Robotics, Concurrent Architecture, Autonomous Robots, Autonomous Driving, Portuguese Robotics Open

Abstract The autonomous robots and unmanned vehicles have been an intensive field of research in the last years for driverless cars and harsh environments exploration. An autonomous robot is a machine that can work in an independent way and subject to its own laws only, a system that can survive in the real-world environment without human intervention. Today many systems are available (e.g. GPS, Computer Vision) that aid the robots to survive, being the biggest challenge put all together and create an agent more intelligent, robust and reliable.

The Department of Electronics, Telecommunications and Informatics of University of Aveiro in the last years has worked in the autonomous driving area. One point of this work has been the developing of autonomous vehicles for participation in the autonomous driving inside competition of the Portuguese Robotics Open, where it has participated since 2001.

The high level software that controls the actual vehicle is based on a sequential structure that turns maintenance and integration of new modules in a complex task. Currently, there are modules to carry out several basic tasks, namely, image perception and integration, "world" representation, and creation of trajectory plans

The aim of this work is the reorganization of the existent high-level software, following a modular and concurrent paradigm. The new software organization makes it easier to replace software modules and to add new functionalities, enhancing team work development and maintenance. The new structure is based on the existence of a central shared memory, like a blackboard, where the modules collect data that they need as well as publish produced data.

This new architectural approach has been implemented in the ROTA robot and it was tested in the 2010 Portuguese Robotics Open (where it ranked 3rd). The proposed architecture links several existing systems and has as strongest points modularity and extensibility in concurrent environment with distributed information.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Autonomous Robots and Autonomous Driving

Robots are synthetic or/and mechanical creatures "that perform tasks by manipulating the physical world"[1]. With embedded computational intelligence, they are "systems with capabilities far exceeding the core components alone"[2, 3]. There is other definition in literature sometimes more embracing, "Robotics is the intelligent connection of perception to action (..) the intelligent connection of perception to action replaces sensing by perception and software by intelligent software"[4] or "a robot is a multi-disciplinary engineering device"[5]. Few years ago Robotics Industry Association (RIA) defined a robot as a "re-programmable, multifunctional, manipulator designed to move material, parts, tools, or specialized devices through variable programmed motions for the performance of a variety of tasks" (Jablonski and Posey 1985), a concept that is almost for "assembly robots", workers that operates in factories.

Nowadays these "beings", created to work inside buildings, start to go everywhere and perform the most varied tasks in wide areas, some of them unachievable by humans. They bring new challenges and requirements specially to work in the real-world (human-centered and life-like robotics) with the less human intervention as possible, introducing autonomy and intelligence[6, 7].

An autonomous robot is a machine that can work in an independent way and subject to its own laws only, a system that can survive in the real-world environment. Thus a robot must have sensors and processing ability that emulates some aspects of cognition and actuators [8, 9]. The sensors retain information about surrounding world, commonly uses computer vision supported by cameras or/and laser range finders, but every sensor that catch important data is an asset. All this data is processed to generate an output that normally gives a motion to some robot element(s).

Some robots have locomotion capabilities, based on caterpillars, wheels, legs, among others, thus enabling the ability to move from one place to another in the environment.[10, 3, 1] The autonomous driving consists in bringing autonomy to mobile robots, providing not only ability to perform actions in the environment but also for robot displacement.

Hence, due to the wide robotics area we can define it as "a mechanism which is able to move and react to its environment"[6].

1.2 Portuguese Robotics Open

In 2001 took place the first edition of the Portuguese Robotics Open, a yearly event, which aims promotion of science and technology through robotic competitions. Initially organized by five universities, University of Aveiro, University of Minho, University of Porto, University of Coimbra and Technical University of Lisbon, is now on the responsibility of the Portuguese Robotics Society. This event has an international scientific meeting and various competitions divided in junior and senior classes.

1.2.1 Junior Class

The junior league includes three leagues, search and rescue, dance and small size soccer.

Search and Rescue

"The competition of Search and Rescue Junior is closely related with the RoboCup Junior Rescue (Rescue RCJ). In this competition mobile robots are used to quickly and accurately identify victims in disaster scenarios that are recreated artificially. These scenarios will range in complexity from line-following on a flat surface through paths with obstacles, line breaks and slopes, to reach an area where the victims are randomly placed in open terrain".[11, 12, 13]

Dance

"The dance competition follows the RoboCup official rules and consists in the realization of a choreography in which one or more robots "dance" to the rhythm of music, being evaluated by a panel of experts in Robotics and Dance. From the point of view of programming, this competition is very little demanding. Nevertheless, the final result, which is the combination of the movement of the robots with the music along with the imagination that is put on some choreographies, achieves good levels of artistic beauty". [11, 12, 13]

Soccer

"This competition is based on a two by two fully autonomous robots, filled with sensors and who's limit dimensions are 22cm, playing soccer. An infrared emitting ball and two different sized soccer fields, with different complexity in the programming level, are the remaining subjects for this exciting competition filled with soccer strategies and goals. The soil of the fields is green and the goals are colored blue and yellow so that robots may indentify them. On the simplest field version there are no sidelines or goal lines and it's allowed to play with the protection walls. The second filed version has sidelines and goal lines and ball is not allowed to leave from inside those lines". [11, 12, 13]

1.2.2 Senior Class

The senior competition includes middle size soccer and autonomous driving.

Middle Size Soccer

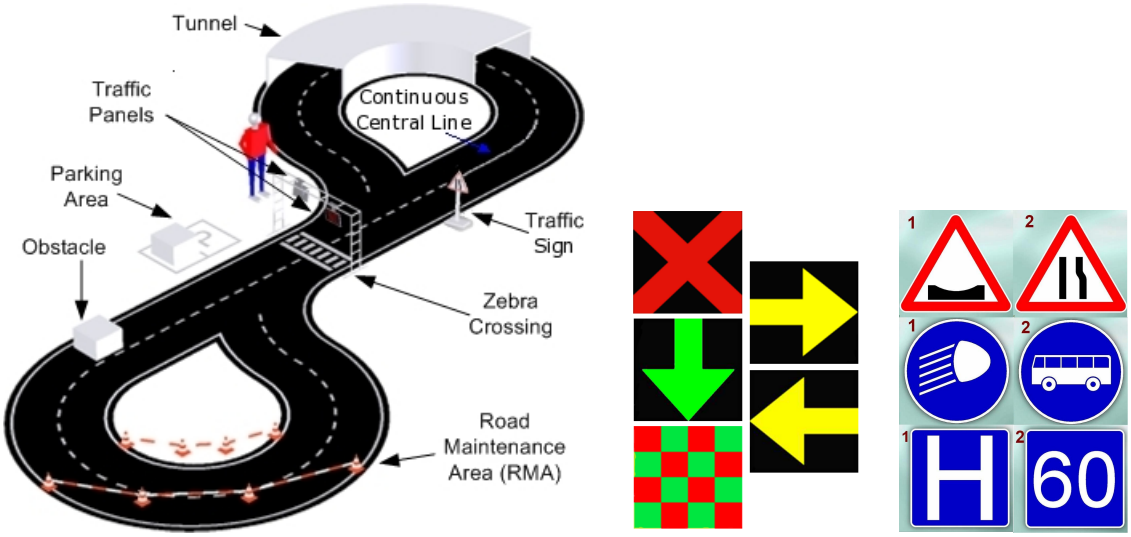
"The Middle Size League (MSL) is an official RoboCup league. Middle-sized robots, of no more than 50 cm diameter, play soccer in teams of up to 5 robots on a field the size of 12x18

metres. Matches are divided in 15-minute halves. All sensors are on-board. Robots can use wireless networking to communicate. Two teams with (typically) 5 robots than can have up to 80 cm height, 50 cm in diameter and 40 Kg of weight, challenge each other in a football field . This field is similar to the human one which has 11 players, but with a smaller size (18m x 12m)". [11, 12, 13]

Autonomous driving competition

The Autonomous driving competition is directly related to the work of this thesis and so, is described in more detail. In this competition a road track is recreated. The track is 8 shaped with two lanes and a fixed crosswalk, above which a traffic light panel is placed, as depicted in figure 1.1a. The road is a black carpet delimited by two continuous white lines and a central one that is dashed. The crosswalk and both lines are made of white tape. The Light signs are: STOP (red), FRONT (green), CHESS (red and green chess) and RIGHT and LEFT (both yellow) as shown in the figure 1.1b. Also, there are dynamic elements, i.e., scenario objects that change their positions along the competition: two obstacles (covered of green carpet), which occupy almost one lane; a tunnel, where the entry and exit vertical edges and its interior walls are white colored; a roadworking area, delimited by maintenance cones, which creates a detour from the original route of the track. The cones are orange and white color (similar to the real ones) and each pair of consecutive cones is connected by red and white tape

There are also 6 traffic signs placed outside the track, along it. The exact positioning of those signs is unknown in advance, and the robot is awarded with additional points whenever it detects and correctly identifies a sign. The set of signs used in competition includes: two warning signs (triangular shape), two mandatory signs (round shape), and the other two are information or services signs (square shape) as depicted in the figure 1.1c. One of this traffic signs identifies a reserved lane that is also defined by a central continuous line as shows on left of figure 1.1a.



(a) The competition track (b) Traffic lights (c) Traffic signs

Figure 1.1: Competition elements [14, 15]

The event is divided in 3 stages, during 3 days (each one per day) along which difficult increases.

The first stage is a speed race with two laps on the track. No elements are placed on the track and the traffic lights are only used to start and end the race. The car is placed at the beginning of the zebra cross and starts running when traffic light change from stop sign to the "follow along" one. During the first three half laps the car can follow any direction (ignoring traffic lights indications) and on the last lap (fourth half lap) traffic lights have the stop sign and the car has to stop.

The second stage has some additional difficulties. Every time the car approaches the zebra cross it has to obey the displayed traffic light signal: stop, go straight, turn left, and turn right. The track now contains an obstacle, whose location is unknown in advance, and the car has to avoid it. In the end of 2 laps and after a light panel indication the car must perform a parking maneuver, choosing one of the two places available.

The third stage has another two challenges, the working area and the tunnel. The tunnel introduces a luminosity problem because there is no light inside it. In the working area the car has to drive using the maintenance cones. In the end of this last stage the car has also to perform a parking maneuver. Since one of the two places is occupied by an obstacle, the robot has to identify the free place and park in it.

This environment is indoors and with restrictions in colors and shapes, i.e. a controlled-environment that simplifies some hard problems of the real-world, like the sun and road wear. More details can be found in the Rules and Technical Specifications of the competition [14].

1.3 The Rota Platforms

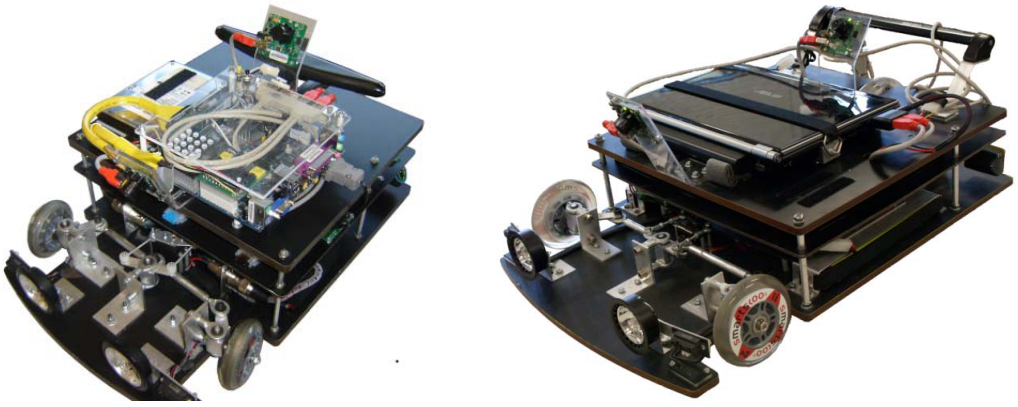
As stated before, DETI/IEETA participates in the Autonomous Driving Competition since the very beginning, in 2001. In 2006, the competition organizers decided to change the track configuration to the one described in the previous section. By that time, DETI/IEETA also decided to build a new vehicle platform. It was called ROTA (RObô Triciclo de Aveiro – Tricycle Robot from Aveiro) figure 1.2a and was developed in the context of a master thesis work [16].

Two years later, in 2008, a new version of the vehicle was developed. It was called RatoZinguer figure 1.2b and the main motivation for its construction was the need to overcome some drawbacks of the previous one. Since the global structure of both platforms is very similar, the next sections only depict the main differences between them.

Mechanical layer

Both robots have the same mechanical layer, 3 wheels (tricycle), one for traction and the other two for the Ackermann steering[1]. The energy is supplied by two batteries, one giving power to the logic layer and another to the traction motor (in the first version batteries also provides energy to motherboard). Cutting off the power to the traction motor puts the rear wheel in free moving, which is quite helpful during development, since the vehicle can be pushed by hand while all the other functionalities can be analyzed.

[1] Ackermann Steering combines two steered front wheels controlled by server motor [3, 16]



(a) ROTA2006 (b) RatoZinguer

Figure 1.2: ROTA Project cars platfrorms

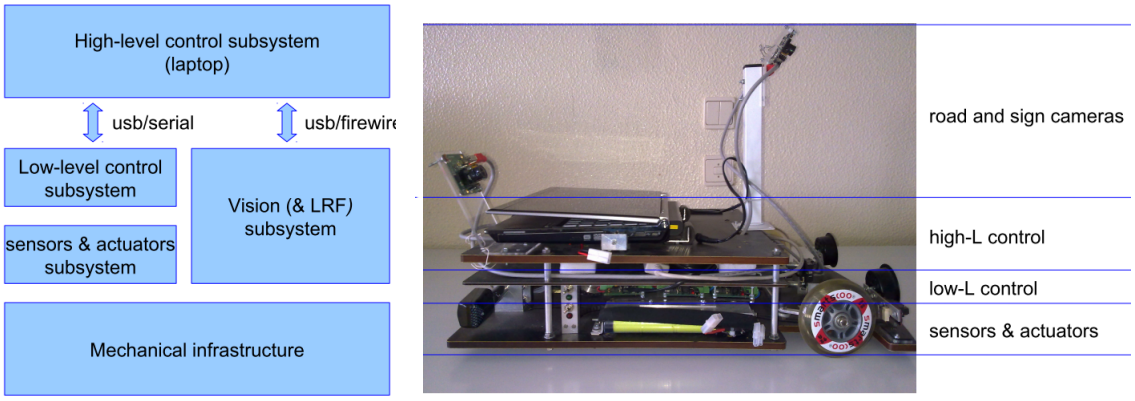


Figure 1.3: Global Architecture

Because the carpet sometimes has imperfections and the car also has a low profile, the ride height was increased in the second version of the platform. This prevents the car stuck or carpet break. The new platform has also new batteries, smaller and light.

Low level layer

The low level layer is composed of a set of nodes that communicates by a through a CAN (Controller Area Network) bus. All the orders for actuators and data from sensors are transmitted to the PC by a gateway with Universal Serial Bus (USB) connection using the network like depicted in figure 1.4.

Vision subsystem

The cars have a camera in the front and other in the rear, being the first used to percept the road and the latter used to detect and identify the traffic signs. In the first platform (ROTA), the front camera is very near to the floor, having an angle closest to 0º to the road, which makes it more susceptible to light interferences and distance distortions. This year the

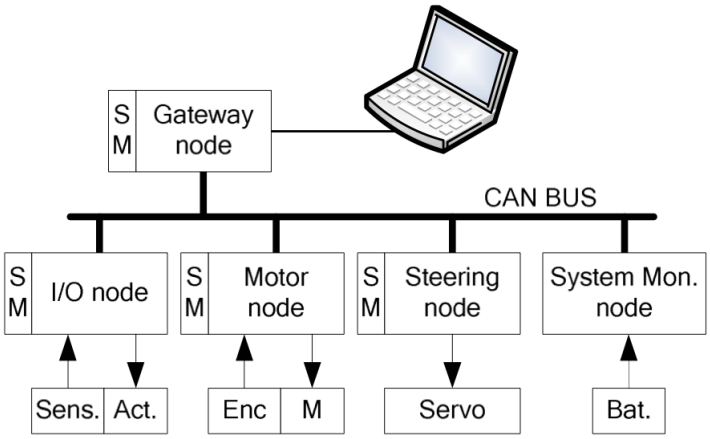
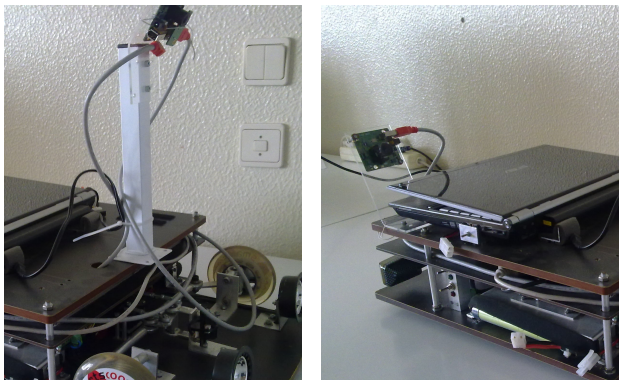


Figure 1.4: Architecture of the base on each car. [15]

new signs are placed at the curb on the track right side and, consequently also on the right side of the car, outside of the field of view of both cameras.



(a) Front camera (b) Rear camera

Figure 1.5: New positions for the cameras, the original is depicted in figure 1.2b

To solve these two issues the mounting positions of both cameras has been changed (see figure 1.5a). The "road camera" is now mounted in a higher position and thus the angle with the road has also increased, which reduces image distortion and potentially the environment light interference. The rear camera has to be placed on the right side of the car to catch the new vertical signs as the figure 1.5b shows. The opening of the rear camera lens was increased in order to have adequate field of view for both the signs, the traffic lights and the new ones along the right side of the track.

High level control

The ROTA vehicle has on top a motherboard for high level processing but this requires an external monitor, keyboard and mouse, which proved to be quite cumbersome in a competition

environment. This gives enough computation power but also consumes energy from the main batteries reducing the overall autonomy.

To avoid this constraint the new platform (RatoZinguer) was designed in order to use a laptop that provides all the hardware to work as well as its own battery. There is also some difference in the computation power as can be seen in the table 1.1

	ROTA	RatoZinguer
Processor	1GHz	Core 2 Duo 2.2Ghz
RAM	256Mb	2Gb

Table 1.1: ROTA versus RatoZinger hardware specifications

1.4 Motivation

The autonomous driving is a great challenge that involves different areas, such as computer vision, data fusion, artificial intelligence, usually in form of behaviors, skills or schemas, and robot control. A common challenge is the need to achieve the best performance always with real-time implications, which makes the best solutions to some problems of not easy (or even impossible) implementation. This area of robotics takes big part in new discoveries, like space and seabed exploration, nuclear works, medicine robots, rescue activities, et al. We need such technology to explore places where the humans nowadays cannot reach, and to do tasks that put their lives in danger or for which they do not have the required precision. Bekey [2] gives the example of the space travel. Going to mars takes more than 1 year and another to back and the planet don't have means to support human life which makes the travel impossible at present (or even in the next years). However, robots can do the job for us. Also the humans naturally make mistakes and in such journey or/and task it cannot happen at all. Avoiding failures is another big challenge.

The control software of an autonomous driving vehicle is a complex structure, composed of several modules, in order to cover the different areas referred above. To deal with such complexity the choice of the software architecture plays a central role. This software architecture can make easy or difficult to insert new modules, to change the existing ones, or to find and fix problems. In the ROTA project the software designed in the last years is now becoming complex to maintain and evolve, and the actual architecture starts to be deprecated. The software of the robot consists of a single main program and several individual modules. Figure 1.6 represents a state machine for complete the last stage of the Portuguese Robotics Open competition, where each one of this state contains a sequence of actions. For example: grab image, process image, get data from sensors, process command to actuators and send command to the actuators. It is a difficult task the implementation of new behaviors and the updating of the current ones. Thus, a new software architecture is required.

Other point which already has been thought for some time ago is the creation of a pit box and also a remote control that gives a better support in developing environment but in competition too.

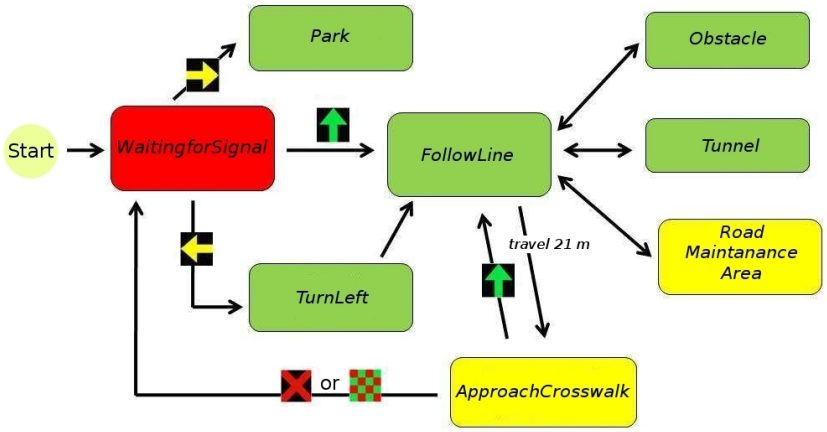


Figure 1.6: State Machine of the old ROTA software, this is where every part of the software is combined to create the agent intelligence. [15]

1.5 Goals

As stated in the previous section, the existing software architecture at the beginning of this work started to be deprecated. The project was started with a monolithic software approach, an unique application responsible not just for a particular task, but can perform every step needed to complete the goals. The project has few years and a faster growing took place in the last two, where new modules were created but the main structure of the software (its architecture) do not suffered any relevant change. Even, there were developed modules, like the path planning one, that were not used due to difficulties in incorporating them in the existing software. Thus, the main goal of this work was to design and implement a new software architecture that overcomes the drawbacks of the existing one.

It was a requirement that the new architecture should be adapted from the one used in the Cooperative Autonomous Mobile roBots with Advanced Distributed Architecture (CAMBADA) project. A good reason supported this decision. The CAMBADA project belongs to the same research group, Actividade Transversal em Robótica Inteligente (ATRI), and is responsible for the development of a team of robotic soccer players. The software architecture used in the robots proved to be a good solution and seemed to be adaptable to the ROTA vehicle. This also allows the sharing of code between projects, which potentiate the saving of development time.

The new architecture unfolds around a central data store, containing information items. Different processes play different roles, producing or consuming items. Part of the data store can be shared between different computer nodes. This allows for the communication between vehicles and between a vehicle and a *Pit Box*.

The new architecture should incorporate as much as possible all the existing modules. For that, these modules may need some adjustments, in which cases they have to be done during this work.

Other constraint on the work flow is the deadline for autonomous driving competition which took place in Leiria between 24 and 28 of March. The majority of the test in the project are performed in the lab that almost all the time has the same scenario. This event

gives an opportunity to test the robot in a track where surrounding environment is unknown and all the elements can be tested together. Thus the architecture should be well defined by that time and implemented so that the event can be used as a case of study.

The software architecture should be very solid and so simple. Some design specifications and requirements are describe in the list bellow.

Reactivity to the environment
The robot should be react to sudden changes in the environment, and thus it needs to promptly respond to world events most of the times in very narrow time windows.

Intelligent behavior
The robot behavior in response to environment events should be made based on common sense rules to exhibit intelligent behavior. Reaction to stimuli must be aware of objectives of its main goal.

Multiple sensor integration
The platform has several sensors with limited accuracy and reliability. The software should process and fuse data produced by different sensors in order to take decisions based on a more reliable source of information.

Resolving of multiple goals
With multiple goals some situations may cause conflicts in concurrent actions or behaviors. The control system should provide means to deal with those multiple goals.

Robustness
Uncertainties of measurements produce unexpected events and may produce wrong driving behaviors. The robot must handle imperfect inputs and sudden malfunctions.

Reliability
The robot should recover from failure continuing, if possible, to work (only if it's not a crucial error that gives malfunctions). Even when a critical error occurs, it should be possible to restart the system and try to continue the normal operation.

Programmability
The project is focused on the competition in the Portuguese Robotics Open but the robot should be easily re-programmed to execute different goals, instead of only one precise task.

Modularity
All the robot software should be divided in smaller subsystems, also called modules, that helps to incrementally add new features and update existent ones. The maintenance is improved as well as the debugging.

Flexibility
The robot is developed taking the autonomous driving competition rules and specifications. However, the research is a key point of the project, and hence the software structure has to be very flexible in order to support continuous changes required by different research experiments.

Expandability
The main aim of the project is the research in autonomous driving area. Hence, the architecture should be easily expansible to build, integrate (incrementally) and test new systems.

Adaptability
The control software as all the architecture must encompasses world changes. These events could be very fast and unpredictable but the system should be adaptable in order to act smoothly but rapidly.

Global reasoning
High level software reasoning should have consciousness of the overall situation and take into account errors from sensing misinterpretation of the sensory data and to fuse the partial available information.

Asynchronously
The hardware car platform is mainly composed of two systems, the CAN network and the Laptop that has asynchronous communication. The architecture should support asynchronous communication and hide it the most possible to programmers.

1.6 Thesis outline

The remaining document is structured in the next chapters:

Chapter 2 gives an overview of available robot architectures in the literature. It presents the paradigms for robot control and the research in the robot architectures from the last years as well as some weakness and strengths of each one.

Chapter 3 presents the proposed software architecture for the high level subsystem. It gives an overview of the concurrent architecture and of the modules that are used on it. This chapter also briefly describes the communication between high and low-level and a pitbox approach for the vehicle as part of a distributed system.

Chapter 4 presents all the details of the modules that belong to the Vision System. The actual platform has two cameras and this is the main source of information to feed the data models. The vision system software is divided in 3 elements, image capture, preprocessing and analysis.

Chapter 5 presents results of the proposed architecture. This depicts some tests and results performed to the implemented modules.

Chapter 6 presents conclusions about this work, as well, several topics for future work.

Chapter 2

Robot Architectures

Even assuming there was a requirement to adapt the CAMBADA software architecture, it was decided to consult the literature in order to analyze which architecture styles are usually used, evaluating their weaknesses and strengths.

This chapter describes the survey done giving a overview of available robot architectures in the literature, and presenting the paradigms for robot control. The former represents how the robot software may be structured while the latter shows different approaches to create a robot control system.

2.1 Robot architecture

"An architecture describes a set of architectural components and how they interact"[17]. Other authors has the same opinion, define architecture as "the abstract design of a class of agents: the set of structural components in which perception, reasoning, and action occur, the specific functionality and interface of each component, and the interconnection topology among components" [18]. Usually the term robot architecture is used for two distinct concepts, the Architectural structure and the Architecture Style. The Architectural Structure refers to the system division and how the different parts interact and corresponds to the structure of the robot system represented informally by the modeling languages like Unified Modeling Language (UML). In contrast, Architecture Style defines some computational concepts in implementations, like process communications as publish-subscribe or client-server [7].

An architecture provides a good organization of a system, a good detail can help the engineer to make choices among design alternatives. However, in addition to providing structure, it imposes constraints on the way the control problem can be solved" [19, 20].

Also requirements are very close to software engineering, where the biggest difference is the inherited for real-time constraint that is not usually the main concern of common software, which only takes in account the human interaction to input and output information. The requirements to this project are described in section 1.5. For example the robot interacts asynchronously the surrounding environment in a real time and the different tasks have different temporal scopes, from milliseconds e.g. avoid obstacles to minutes e.g. task planning.

2.2 Robot Control Paradigm

The robot control defines how much the robot "thinks" to take the control commands. In other words describes the organization of the control parts in order to produce actions from the sensors readings. There is a wide range of options for robot control, which are usually grouped in four different classes: reactive, deliberative, hybrid and behaviour-based.

2.2.1 Reactive paradigm, Don't think, (Re)Act

The "reactive control is a technique for tightly coupling perception and action, typically in the context of motor behaviors, to produce timely robotic response in dynamic and unstructured worlds." [9]. It is equivalent to the biologic notion of stimulus-response and do not store any kind of state. So, it provides a very faster response in real-time and unstructured world environment [21] since it is composed of concurrent and preprogrammed behaviors based on a very simple computational processes. [22]. The simplicity is due to the lack of models, because usually work with models are usually a very hard computational task and takes much time to complete.. The reactive control is similar to mathematical functions, since in "these reactive behaviors sensing is directly associated with acting" [22].

Data is received from the sensors and, almost directly, an output is produced, that is automatically executed by the actuators as depicted in the figure 2.1. The reactive control "is modulated by attention and determined by intention" [9].

Figure 2.1: Reactive Paradigm

The lack of complex models turn this paradigm very simple and similar to some creatures in nature, like insects, that has very limited (if any) ability to store information and are majority reactive [23], and obviously did not produce the more intelligent or optimized tasks but very responsive to highly stochastic environments.

Some examples of reactive behaviors are: *Follow a person*, *navigate through a doorway*, *stop /avoid when the robot is too close to an obstacle*, *etc.*

2.2.2 Deliberative paradigm, Think, them Act

The Deliberative robot is composed of premeditated actions that are based on the world state created by internal knowledge acquired from environment sensing.

Usually the system creates a sequence of actions to achieve the goal, called plan and then perform the desired action. Computationally, planning has to test all the possibilities, taking into account the constraints. Therefore it is a very complex decision-making process to perform a sequence of actions. Thus, to support planning, internal world models are essential

but creating such an abstraction may be a non-trivial task. Maintaining it as accurate as possible is also a complex task. This paradigm is centered in knowledge, as the figure 2.2 shows.

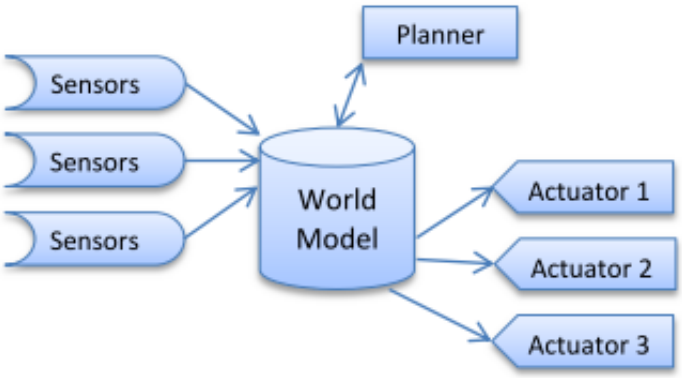


Figure 2.2: Deliberative Paradigm

In a dynamic and noisy world (and other external interferences) it is impossible to make use of purely deliberative control [24]. Because of the time required to complete the planning, deliberative robots cannot achieve a quickly response to events coming from world dynamic like moving obstacles, thus the collision avoidance is not guaranteed. Nowadays, even robots that are created to very specific situations (Situated Robots), are not purely deliberative[7].

The storage of models and other types of information requires a symbolic representation and the abstraction is a signal of Intelligence. "It could be argued that performing this abstraction (perception) for AI programs is merely the normal reductionist use of abstraction common in all good science. The abstraction reduces the input data so that the program experiences the same perceptual world as humans" [23].

This paradigm is more intelligent than the reactive one and can deal with very complex tasks in structured worlds due to the planning, e.g. the robot search the shorter path to move from one point to another and then execute it. However, due to be planning-based it cannot be quickly responsive to environment changes and then can be a bad choice to dynamic environments.

2.2.3 Hybrid paradigm, Think and Act concurrently

The hybrid control joins the best features from reactive and deliberative paradigms, i.e. make use of premeditated motion and at the same time use reactive behaviors.

The majority of the hybrid architectures are composed of three blocks as depicted in figure 2.3: the deliberative component (organization or planning) *thinks* the best way to achieve the objectives, a reactive or execution component that constantly sense the world and usually implements tasks very close to the human instinct or reaction, like obstacle avoidance. The intermediate component is connected to world model and sensors but can be almost reactive, using directly the sensing data to produce an output or being more deliberative. This commonly uses a three-layer architecture [25] and is an example of the "Principle of increasing intelligence with decreasing precision".

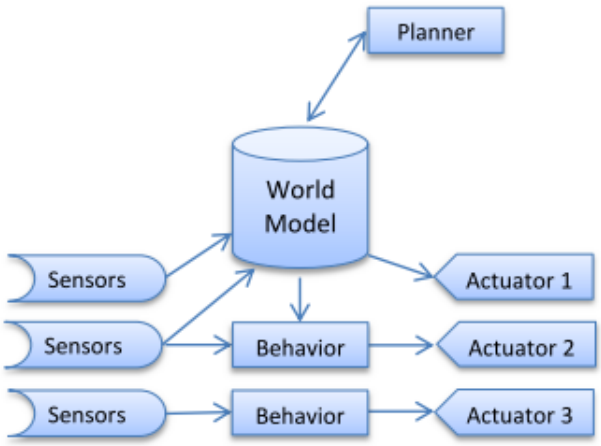


Figure 2.3: Hybrid Paradigm

Since there is a reactive part the robot can be very responsive to environments events (changes) while execute the predicted reasoning. However, the reactive and deliberative can be contradictory. For instance, the deliberative piece gives the order *go front*, while the reactive one, because a sensor have detected an obstacle, gives the order *avoid obstacle*. Thus, a key point here is to resolve this issue, that is, select the action that increases the probability of achieving the goal.

2.2.4 Behavior-Based paradigm, Think the way you Act

Accordingly to the behaviorist school of psychology, *Behavior is a reaction to a stimulus*. From the stand point of the programmer, behavior is the aggregation of control modules that lets the robot achieve and maintain goals. From the outside the behavior is a type of action between the robot and the surrounding world [9, 26]. The robot actions are not implicitly achieved from robot or the environment, but rather as a result of its behaviors interaction. This has little difference from the notion of behavior presented in Reactive Control 2.2.1, whereas this behaviors use simple rules and no models or states are kept, while the behavior-based behaviors can store states and other information, enabling reasoning.

All the behaviors together in a network can recreate a model and enable the reasoning and hence planning and learning. However, behaviors should be simple as possible to maintain the ability to be reactive to environment events. Such network can be implemented using layers where the base (composed by the simplest ones) is similar to pure reactive, e.g Collision-Avoidance and the top supports the reason (the more complex). Unlike the reactive paradigm that reaches a good performance when the world dynamic is not easily extracted or predicted, the Behavior based try to learn and avoid mistakes from past looking into the future actions.

An improvement in behavior representation from reactive ones is the notion of "Activation condition" and "stimuli". The former enables the behavior and the latter acts like stimulus in reactive-behavior to select and produce an action. More than one behavior can be active at the same time (concurrency), whence behavior outputs must pass through an action selection module, as depicted in figure 2.4. There is no perfect algorithm for filtering actions, but a good selection can be determinant in the control performance.

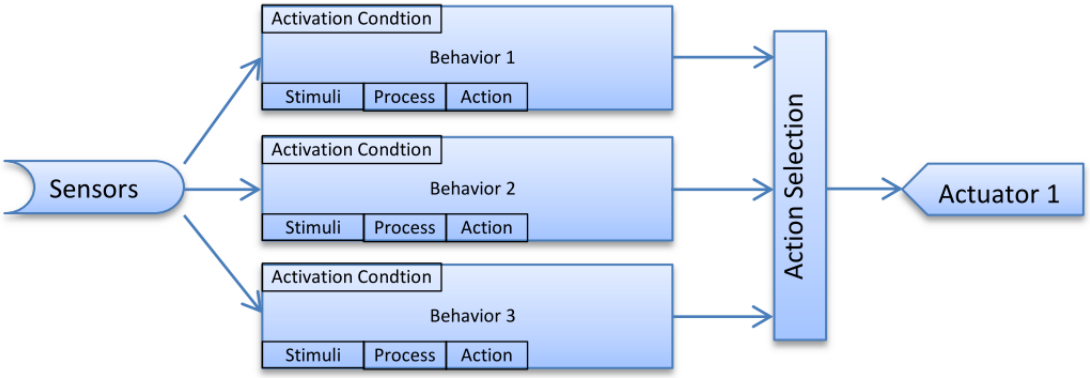


Figure 2.4: Behavior Based

Some authors grouped behaviors in three broad categories, namely reflexive behaviors, reactive behaviors and conscious behaviors. Reactive and reflexive behaviors are a bit different. While the reactive has a simple and very fast reasoning the reflexive is similar to a "hardwired" connection between sensors and actuators, e.g. knee-jerk reaction when a doctor hammers your knee [27, 28].

2.3 System architectures

The raw material for robots has been developed over the last 50 years, and the need to joint all these pieces requires an architecture. The next pages will depict several architectures, giving an overview of the most cited in literature.

2.3.1 SPA (Sense Plan Act)

In the later 60s Nils J. Nilsson at SRI[1] published a paper [29] with one of the first approaches for a robot architecture. The main objective was to build Shakey 2.5, a robot with a camera, range finder and bump sensors, to move in a real-dynamic world and to achieve some tasks. Such abilities can be found individually in robots or in some intelligent programs at that time and the author grouped them in 3 classes, Problem-solving, Modeling and Perception.

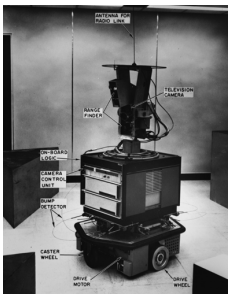


Figure 2.5: Shakey Robot [30]

The problem solving creates a sequence of primitives that lead the robot to accomplish given goals. The Modeling has the knowledge about the effects of actions and the world states that is updated by the own actions or sensor data. Finally, the Perception that, as the name implies, involves camera images and sensors processing to produce a robot understandable data and feed world, in other words retrieve relevant information from data. So, Perception sense the environment (Sense) and the Model stores the data in models (Modeling). The Problem Solving encompasses achieving a solution as well as to execute it, in other words, plan what to do, how to do and control the action in order to complete goals. Sense-Plan-Act (SPA) architecture is more cited in literature and always presented with the 5 subsets mentioned above, like depicted in figure 2.6, where Perception and Modeling are parts of the sensing, while Planning, Task Execution and Motor Control are parts of the problem solving Problem Solving one [21, 30].

Each level or functional unit uses only the information produced by the previous one, except the sensing that produce the first data from sensors. This turns into a sequence where each level is executed only when the previous is done.

If one of the functional units takes much time to accomplish results it makes the sequence to be delayed and consequently the robot is slower. The planner, in a real-world with lots of information cannot work in real-time, because usually has heavy processing. The planning can produce a larger set of directives and execute them. In a dynamic-world it is dangerous because no sensing shall be done at the same time in order to be responsive to events.

[1] Stanford Research Institute

Figure 2.6: SPA, traditional Artificial Intelligence robot brains are serial processing units.

The SPA is a deliberative architecture since it produces a plan and then execute it without sensing until the end of the current plan.

2.3.2 Subsumption

SPA architecture has two big problems. The planning can take a long time to process and its execution without sensing in a dynamic world can be dangerous. The expansibility to complex robot of this architecture is very difficult and almost impossible. In the 80s, Brooks proposed the subsumption architecture [21], also called subsumption behavior-based or behavioral robotics. Unlike the SPA (vertical slices) the proposed architecture uses horizontal slices, and each one is usually a state machine that takes input from sensors and outputs to actuators. This state machine is often called behaviors that support the name. The paper refers some requirements and also assumptions.

The robot can achieve multiple goals, but some of them can conflict with each other and to resolve this issue a priority schema is assumed, where highest behaviors subsumes lowest ones 2.7.

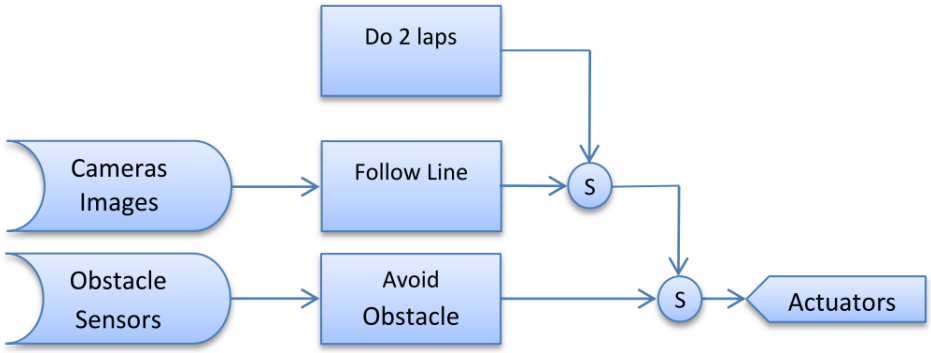


Figure 2.7: Subsumption layers.

A robot that moves in a real-world generally depends on multi-sensors and this introduce various errors, like the reading, transmission, out of range values, domain applicability etc. (A common question in literature is: "Under what precise conditions does the Sobel operator return valid edges?") The robot must make decisions under these conditions of uncertainty and must continue to achieve tasks as long as possible even when something fails. This tolerance ability gives more robustness to the robot. Even when something fails the robot can operate and as long as possible achieve tasks.

A robot can be updated, for example adding more sensors, and consequently more behaviors to use the new data are needed. Even the orderly behaviors can be modified or re-arranged, but in this architecture the goal is always to expand the actual functionalities, increasing the number of behaviors. The idea is to keep components and interfaces simple, being the robot complexity a product of the environment complexity and not of the behaviors by themselves.

2.3.3 SSS

In 1992, Jonathan H. Connell described a new three layer architecture: Servo, subsumption, and symbolic (SSS)[31]. The objective of the project was mapping a floor and then navigates from one office to another with people and some obstacles in the way, using the TJ Robot 2.8.

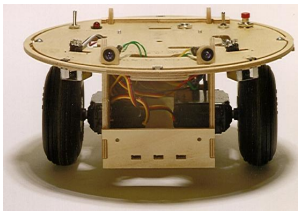


Figure 2.8: Talrik^{TM}Jr. AVR SSS testing

Instead of using just a behavior-based approach, like the subsumption, the control of the robot was structured into three layers (see figure 2.9). At the lowest level, there was a layer (servo layer) that dealt with lower data. It was responsible for keeping the movements smooth, controlling the wheels speed or steering and highest layer that implement the most intelligent part.

Above the servo layer there was the subsumption layer similar to the described above in the subsumption architecture. It was constituted by a set of behaviors that are selected and parameterized by the upward layer. This last layer is referred as symbolic layer and typically use events detected and created by pattern recognitions or some specific situations by the subsumption layer.

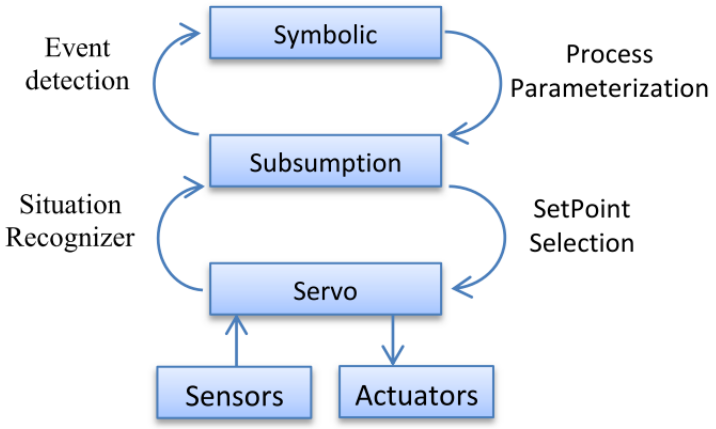


Figure 2.9: Servo, subsumption, and symbolic (SSS) by Jonathan H. Connell

These make usage of 3 different approaches in control system, servo-like, reactive rules and discrete-time symbolic system, but the author referred that "this is not to say a good robot could not be built using just one of these technologies exclusively. We simply believe that certain parts of the problem are most easily handled by different technologies".

This corroborates the Brooks idea, that there is no optimal solution, usually the problems should be divided in simple ones and resolve each one in an optimal way (if it exists). The difference between this architecture and the subsumption one is that Brooks argued that the reductionism applied to data to transform it in symbolic information (used here by the symbolic layer) "will lead us to solving irrelevant problems; interesting as intellectual puzzles, but useless in the long run for creating an artificial being" [32].

2.3.4 ATLANTIS

The architectures presented at this time (in the 90s) were usually based on behaviors that are described by state-machines, supported by the idea that we can describe operators onto the world model and sequence them to perform actions. Erann Gat around the same year (1992), proposed the ATLANTIS [33], acronym of A Three-Layer Architecture for Navigation Through Intricate Situations, an architecture based on action-model instead of the most popular behaviors. Action denotes two different things, the action performed by the robot in the world from the outside viewpoint, but also each action that internally composes it (from inside viewpoint) that is also called operator by the author.

When an operator is triggered, it starts a process (activity) and in the end the operator decision can trigger other operators and/or activities. If these operators network does not have any cycle, then hierarchy can be created where the high-level can initiate low-levels operators and/or activities. From outside viewpoint the robot action can be a sequence of operators and so, activities cannot be observable alone. This blinding encapsulation turns very hard the analysis of the action-model approach.

The ATLANTIS architecture is composed of three main components: the controller, the sequencer, and the deliberator or planner, as is depicted in figure 2.10. Like the other architectures presented before the control layer is responsible for execution, using the control theory applied to a set of activities created by the sequencer and ordered by the executer. The top layer is the planning and a world modeling that creates a plan to be divided in the activities by the sequencer on middle layer. Many environments properties and models are hard to represent in computational information, and, even if is possible, it can be proved that is impossible to do it in real-time. Chapman made reference to a NP problem[2] in her Intractability theorem [34, 35]. Thus, one of the most interesting points on this architecture is that "classical planner should be operated synchronously in conjunction with a reactive control mechanism, and the planner's output should be used to guide the robot's actions but not to control them directly" [33].

2.3.5 AuRA

In the 80s, Ronald C. Arkin at the Georgia Institute of Technology developed, AuRA, another hybrid architecture. In this case the deliberative and the reactive components are completely separated and some features are inspired in biologic knowledge (in books and papers of the author is commonly found neurologic expressions like "long term memory" or

[2]NP-complete in the Computational complexity theory, one of the Mathematic Unsolved Problems

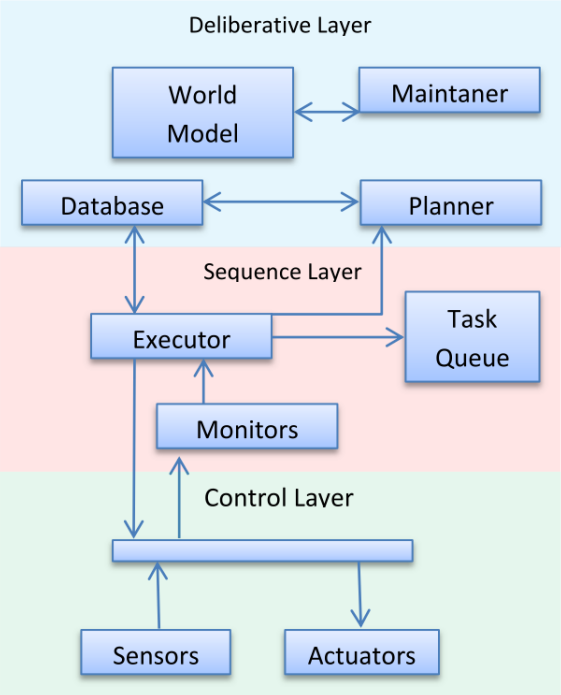


Figure 2.10: ATLANTIS Architecture

"schemes"). "Fortunately, there exist excellent mobile systems (animals), provided by God, that can be studied to yield insights into the mobility problem" [36].

AuRa is divided into 4 layers, as depicted in Figure 2.11. Initially the user gives an order or goal to the Mission Planner (user intentions), and the mission planner using also some planning recognition creates a plan. The Spatial Reasoner makes use of previous knowledge stored in long term memory (e.g. maps) result of learning (spatial learning) or some previous user input (spatial goals). This produces a sequence of actions or behaviors (named by the author as schemas) which are then executed by the Plan Sequencer referred as Pilot or Driver. This sequence by virtue of opportunism can be changed or also by some user indications of mission alterations (e.g. task D becomes more important than C). The first steps are the deliberative layer of the architecture, and then the Schema Controller will execute the sequence of schemas created behind and monitors it which makes the reactive part of the architecture. The actuators controllers can adapt them to perform actions more smoothly. At this level also the user can "drive" the robot, sending orders directly to the controllers. Note that the deliberative layer is only re-done if the robot has a failure.

In 1997, Ronald C. Arkin and Tucker Balch published a review denoted "AuRA: Principles and Practice in Review" and show, in ther opinion, the strengths of the architecture, some of them very useful in development environments: "modularity, flexibility, generalizability and hybridization is the constitute the principal strengths of the Autonomous Robot Architecture" [37].

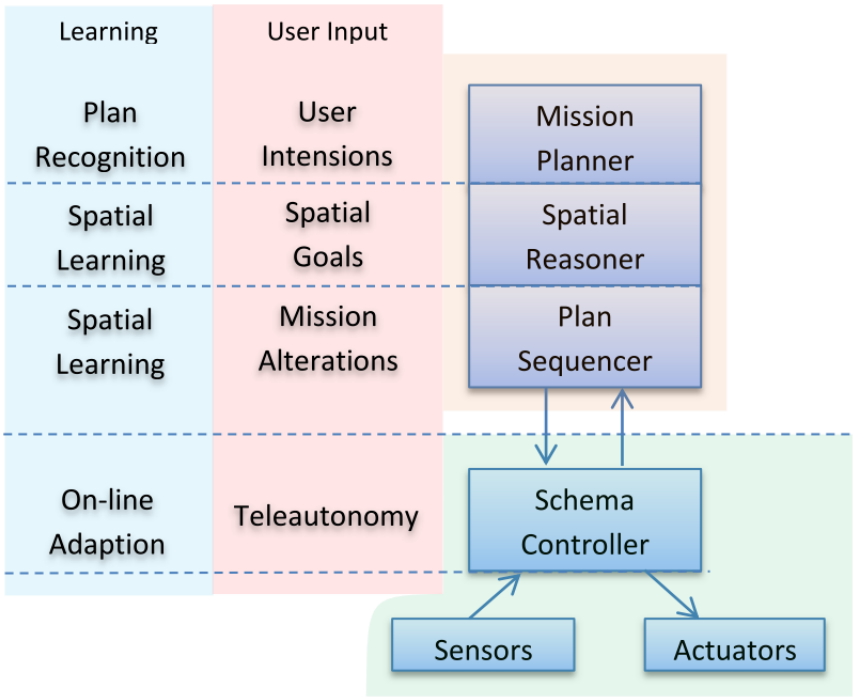


Figure 2.11: Autonomous Robot Architecture

2.3.6 Three Tiered Architecture

The Three-Tier Architecture, also called Three Layer Architecture, was developed by R. James Firby under his PhD thesis [38], but the term 3T (three Tier) only appeared in 96 by Bonasso et al [39]. The system is based on Reactive Action Packages (RAPs), also designed by the author under the PhD thesis. RAPs describes how the robot can achieve a goal as well as the available methods (for different world situations or constraints) to complete assigned tasks. The 3T have 3 distinct layers, the **Deliberative or Planner**, **the Sequencer** and the **Executor**, as shows the figure 2.12.

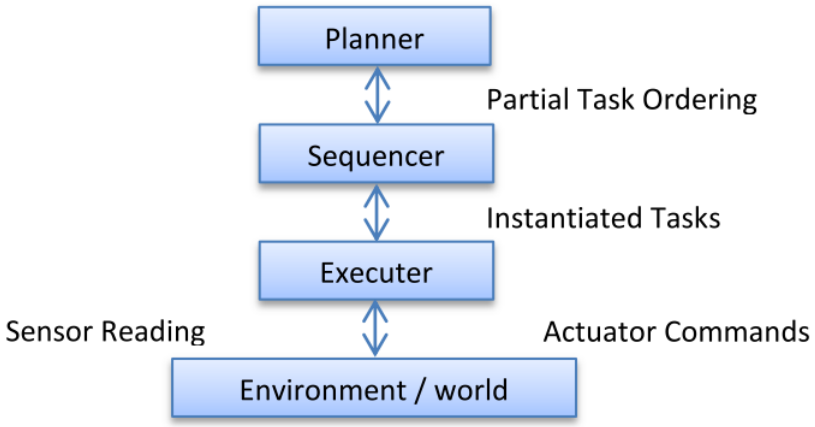


Figure 2.12: Three-Tier (3T) Architecture

The Deliberative works to create a sequence of actions to accomplish a set of objectives in a highest level of abstraction and the result should be the division in simpler and easier tasks. The Sequencer, use the planning result and selects what action from RAP must be enabled. This is also called of "RAP Interpreter" [38, 39] since the RAP can define more than one different form to do a task, depending on the circumstances. A task description is a sequence of steps and each one is a set of basic directives to be executed by the lowest layer by the RAP Executer, or only Executer, in a concurrent environment.

When a robot perform an action the reenforcement is used to give learning ability to the robot. So if the robot select a RAP that is executing and it does not produce the expected behavior, the sequencer is warned. Sequencer can also warn the deliberative part.

2.3.7 Triple-Tower Architecture

The Triple-Towers architecture was borned in the end of 90s, in fact the first reference to "Triple-Tower" is in the book "Artificial intelligence: a new synthesis", but this is proposed as rough, sketchy, and not under any kind of implementation and so needs more elaboration [40]. This is based on the idea of layered architectures, where the first level deals with the data in raw format and the next levels will process the data from the descendent layer and create more abstract information, and so on [40, 41]. Abstraction and reasoning should be concurrently achieved over the time in all system and the number of levels is not specified or limited in any way. The presented architecture is focused on the hierarchy idea and makes this more a conceptual model than an architecture. The hierarchy creates the illusion of three towers, the **Perception or Sensing**, the **Model or World Model** and the **Action or Behavioral Generation** as is depicted in the figure 2.13 and gives Triple-Tower name.

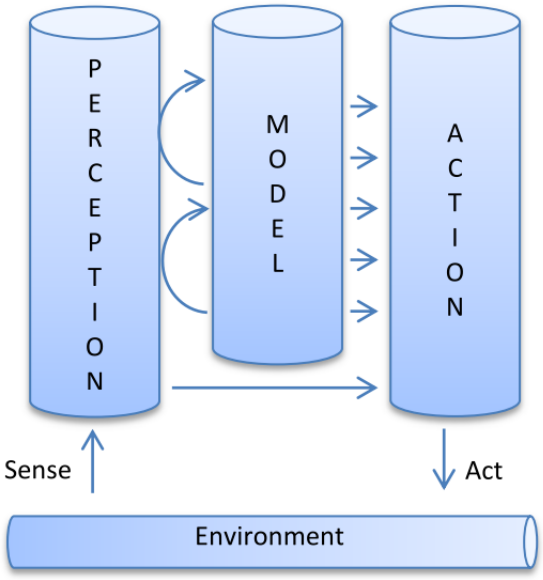
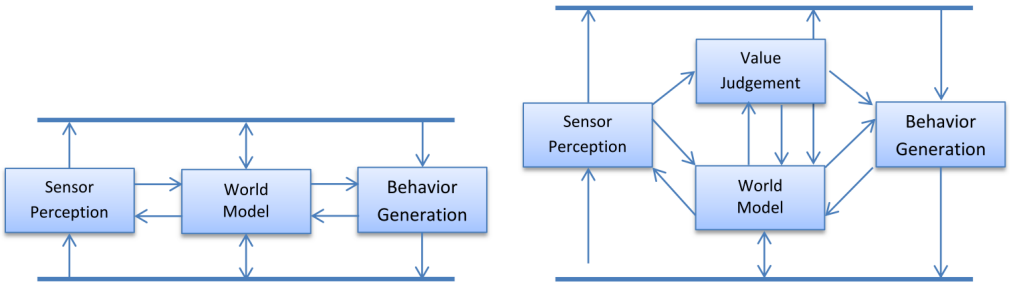


Figure 2.13: Triple-Tower Architecture

The Learning has an important focus in the Artificial Intelligence area and the layered architectures give a better support to it because layers can be created on top of the existing ones and then increase the abstraction that somehow increases intelligence. For intelligence,

sensing, decision-making and acting are essential elements, but a better intelligence includes object and events recognition, and more abstraction that takes planning to another levels [42], and this is implemented over the highest levels.

The 4D-RCS architecture and the RCS-3 and RCS-4[3] paradigms are the basis of the Triple-Tower Model and also a reference model for NASA[4] and military unmanned vehicles [7, 27, 43]. The RCS paradigm defines the functionality of a transversal layer, i.e., a layer covering the three towers, and the two variants, RCS-3 and RCS-4, are depicted in figure 2.14. It defines the functions of the modules at each tower, how they interact and how equivalent modules in adjacent layers interact.[42]. The difference between the two paradigms is the Value Judgment module, present in RCS-4 but not in RCS-3, that act as a layer supervisor. Layers are very similar in terms of architecture, and also communications flow. Thus the main difference is the order of abstraction of reasoning and so we can focus only in the architecture of one layer.



(a) RCS-3 Paradigm (b) RCS-4 Paradigm

Figure 2.14: Real-Time Control System Paradigms

The communications are always between adjacent layer on each tower, but inside one layers the modules from adjacent tower also communicate. This data flows sometimes are a multi interaction, i.e., the action layer can query model and subsequently the model query the action [42].

The spatial and temporal reasoning decreases about an order-of-magnitude at each higher level [42]. So other interesting viewer of the layers hierarchy is depicted in Figure 2.15.

The Servo control works in a high frequency to maintain the motors with the correct angle and velocity on each order passed by the behaviors which runs at a lower frequency. This is temporal but in space the same occurs, the behavior looks more "ahead" than the servo. A planner can create a large plan in space, also thinking in long term and runs at a lowest frequency of the behaviors.

Thus, more abstract layers work at more lower frequency but process a larger window of space and tine.

[3]Real-Time Control System
[4]National Aeronautics and Space Administration

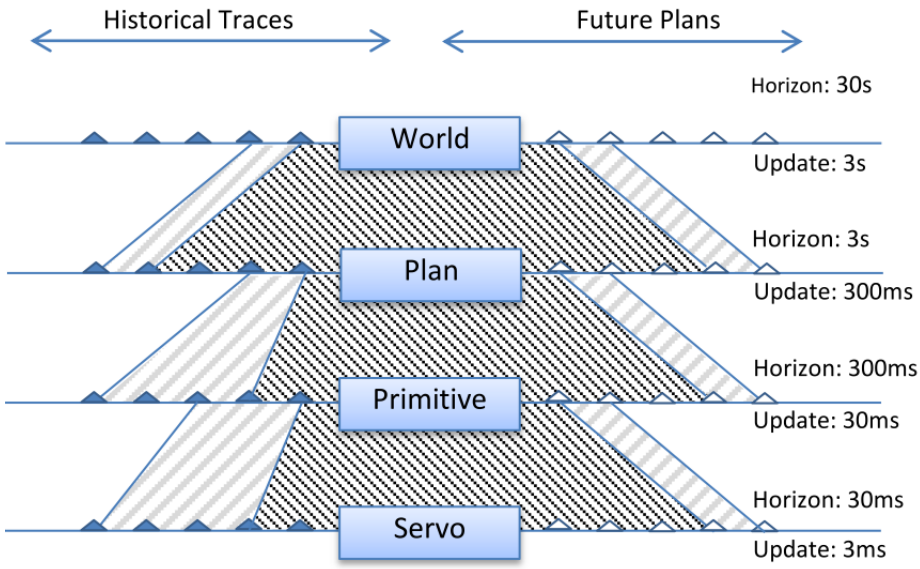


Figure 2.15: Space and Time resolution in each layer

2.4 Conclusion

Over the last 50 years lots of architectures were proposed, created and tested, being many of them successfully re-used, some with minor variations, in different environments and/or tasks. However, most of them were created to an environment type, a kind of "situated architecture". All were tested in a large amount of robots that implement their architecture approach. It can be concluded that both, the reactive and the deliberative control, are useful, but for different environments: the former for the highest stochastic and the other to controlled environments like factories. However, the hybrid solution can be more interesting in the general case, since it can use more or less reactivity/deliberation, while enjoying the advantages of both. The next board shows what is best in each one.

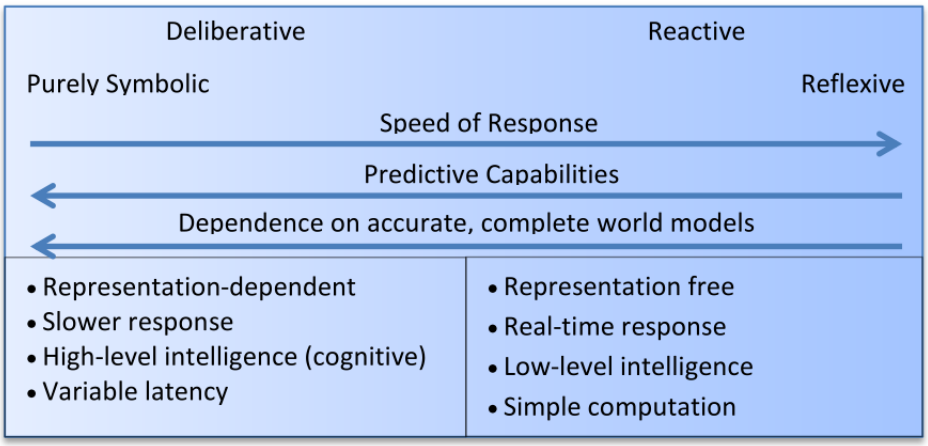


Figure 2.16: Reactive vs Deliberative

The Behavior-based is a design methodology for reactive systems that is focused in the idea of behaviorism, where the behavior is defined in terms of stimulus and response. Despite being a reactive system, it implements intelligence spread by behaviors, unlike the deliberative and hybrid that have a specific deliberative modules in its structure. The strength of Behavior based architectures is the modularity, and the intelligence created by a set of autonomous modules, which, when work together produce intelligence.

The layered architectures also gives give a better modularity, flexibility and expandability since each module / layer has an API that facilitates / makes easier the *block* exchange. The modular architectures give a highest extensibility to the robot and then increases team work productivity (e.g. behaviors can be produced in parallel and also easily exchanged). However, this raise a big issue: when behaviors are executed in a concurrent environment how are them selected to avoid collisions of the orders (output)? Some algorithms can be created but almost to specific situations. Using a priority approach the intersection of the behaviors output can be avoided but it is not guaranteed that the robot produces the more optimized task.

Chapter 3

ROTA Architecture

"The advantages of distributed architectures extend from improved composability, allowing a system to be built by putting together different subsystems, to higher scalability, allowing to add functionality to the system by adding more nodes, more flexibility, allowing to reconfigure the system easily, better maintainability, due to the architecture modularity and easiness of node replacement, and higher reduction of mutual interference, thus offering a strong potential to support reactive behaviors more efficiently. Moreover, distributed architectures may also provide benefits in terms of dependability by creating error-containment regions at the nodes and opening the way for inexpensive spatial replication and fault tolerance" [44].

The former ROTA's high level software sub-system (see figure 1.3) was monolithic, in the sense that there is a single thread of execution that sequentially implements the different tasks. Each execution cycle starts with the road frame acquisition. Then this frame is processed in order to extract navigation information. Next, it is decided what the next actions are and, finally, those orders are sent to the low-level software sub-system. Close to the zebra cross sign camera frame acquisition and processing are also inserted into the sequence. This approach worked nicely, but started to show some drawbacks as the complexity of the software grew up. For example, a track description and a path planning modules were developed, but are not integrated on this software due to the difficulty of integrating new modules.

During this work a new architecture for the high-level software subsystem was developed. It is based in a multi-process, multi-thread approach, using a central data store as the communication framework. As already stated in the Introduction, it is adapted from the software architecture of the CAMBADA's robotic soccer team. Since the CAMBADA's architecture was designed for a team of collaborative robots, it can also be used to allow communication between vehicles or between a vehicle and a *Pit Box*.

Figure 3.1 gives a global overview of the proposed architecture. Each vehicle and the *Pit Box* act as nodes of a distributed system, supported by a wireless network. The communication process, running at each node, guarantees that the shared part of the ROTA Data Base (RtDB) is replicated among all the nodes. For the typical competition, where the vehicle has to be completely autonomous, these communication facilities cannot be exploited. However, during development it can be an important aid.

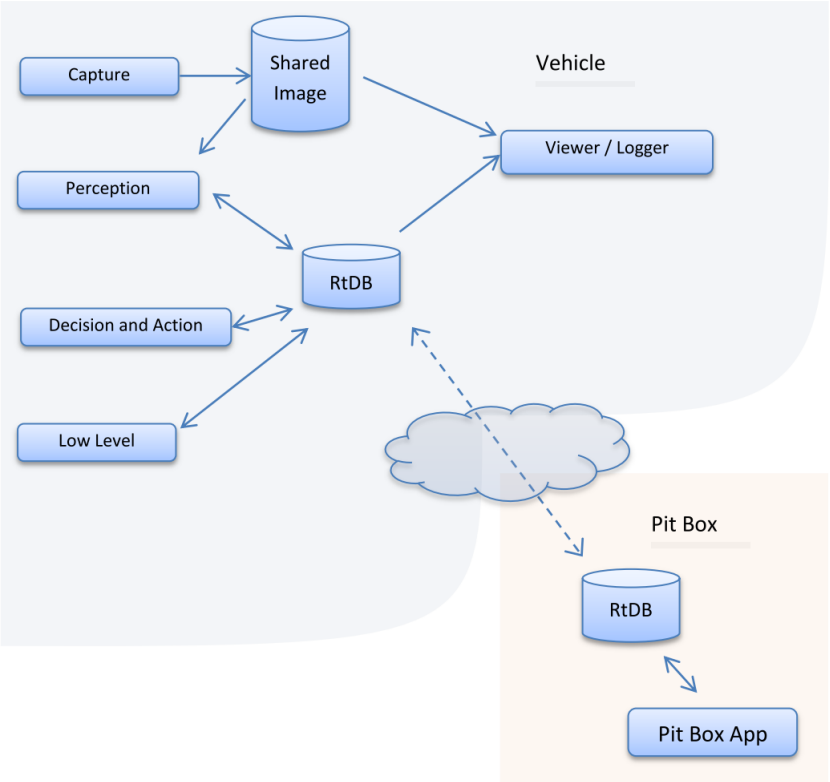


Figure 3.1: ROTA architecture components

3.1 Architecture module

The modules referred in this chapter are structurally similar and can be considered as black boxes. Using a central database all the modules can be structurally modeled (from outside view point) as depicted in the figure 3.2. This definition appears in literature but only for behaviors (see section 2.2.4). Although, extending this concept to all elements increase system optimization and the reasoning is consequently more efficient since it can be enable/disable or use other information to take influence in processing steps.

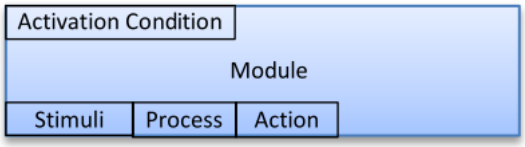


Figure 3.2: Generic descriptor of entities

The four elements that constitute a module are described below.

Activation Condition

The activation condition is used to enable or disable the activity and then processing time can be used to another tasks. In situations where lots of entities are disable it can save batteries power, e.g. disable the light signs camera and/or signs processing when

sings are not present.

Stimuli / Input
The Stimuli can be from two sides, inside stimuli and outside stimuli. The internal stimulus, comes from database and is used to restrict or conditioning the process step, allowing more accurate and rapid results. For example, after an initial search, the lines in roads usually do not move much from one frame to another. The information on map can also be used to anticipate new lines or the end of the existing. The external stimulus is the input data to process, like images or sensors data.

Processing
This is the kernel of each entities. Ti is created to process data and produce an output, in other words this is the algorithm implementation part.

Output / Response
The result of processing step is stored in the database. This is then used by other entities as stimulus, or sent as orders for actuators.

3.2 Rota Data Base

The CAMBADA team from University of Aveiro has carried out some research in cooperation between robots for the middle-size robotic soccer team. The goal is to playing a soccer game in a cooperative environment, which requires coordinated actions and interaction among all the robots. The underlying of CAMBADA architecture is a distributed Real Time Database (RTDB) which keeps all the players with information from all other ones, and about the game in general. This supports dissemination of data between players and also between players and a base station that is able to send indications to the players. Since each node on the distributed database has information about the others, and the base station is also a node, it can show information about each player, acting as a monitoring tool.

The database works with abstract items that are identified by a pair of IDs, one that identifies the node and another that identifies the item inside the node. Each item is just an array of bytes that stores any information that can be shared or information local to the node. The shared data are replicated over the nodes, while the local is used only to inter-process communication within the node itself. The distribution of RTDB shared items is managed by the process, running at every node, called *comm* (figure 3.3). This process replicates the shared part of RTDB over the nodes with temporal information and uses a management system to schedule the traffic and improve the real-time communication and enforce timely updates of the database items, dynamically adapting to the conditions of the communication channel [45].

The RTDB is implemented in Linux OS and uses a shared memory block, which can be accessed concurrently by all the node processes. The available manipulation methods implement a *copy to use* strategy, i.e. instead of blocking items to prevent multiple accesses, they atomically make a copy of the desired item.

The RTDB has a simple API to create, interact and destroy the database. The initialization is performed using the method **DB_init()** and stopped using the method **DB_free()**. There are also two simple methods to interact with items, **DB_put()** and **DB_get()**. **DB_put()** is used to write items into the database. A node can only write its own shared

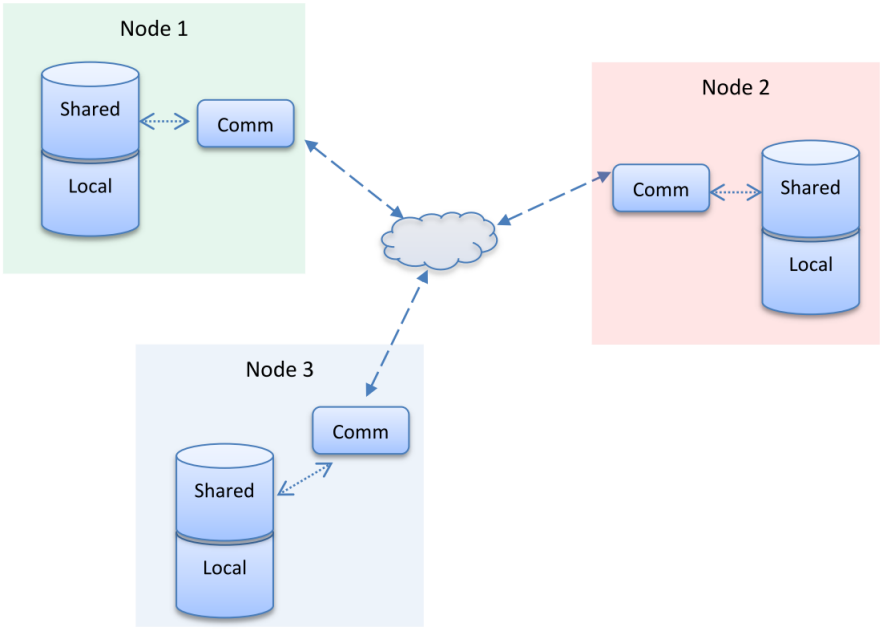


Figure 3.3: RtDB, a distributed database

items, i.e., the ones it produces. The others are written by the already mentioned comm process. **DB_get()** reads items from the database. The C prototypes of the RTDB related function calls are:

int DB_put(int item_id, void *value);

int DB_get(int node_id, int item_id, void *value);

int DB_init (void);

void DB_free (void);

The initialization function of the RTDB uses a low-level configuration file to know what to do. This file describes the nodes in usage, the items at each node, their nature, shared or local, and their size in bytes. At this level, nodes and items are just numbers. At a higher level, a configuration file is used to feed a parser application, which converts a set of C++ class definitions and associated names to the abstract items of the RTDB. The names are converted to C++ macros, which makes the access to the items more user-friendly.

The RTDB definition is abstract enough to be used without any modification in the ROTA architecture. The only differences are the node names, the item names, and the classes associated to these names. However, that is managed through the configuration file and, therefore, the only work to be done in order to use the RTDB in the ROTA architecture is to write the configuration file, based on the required items (appendix A.1), and run the parser application.

3.3 Pit Box

The pit box is proposed as a node that can be an excellent aiding tool, particularly during the development phase. It is composed of an instance of the RTDB, a *comm* process and the *Pit Box* application as depicted in the figure 3.4. The latter is the main component of this node and should be a graphical application, used both to control the vehicle and to display information retrieved from the vehicle.

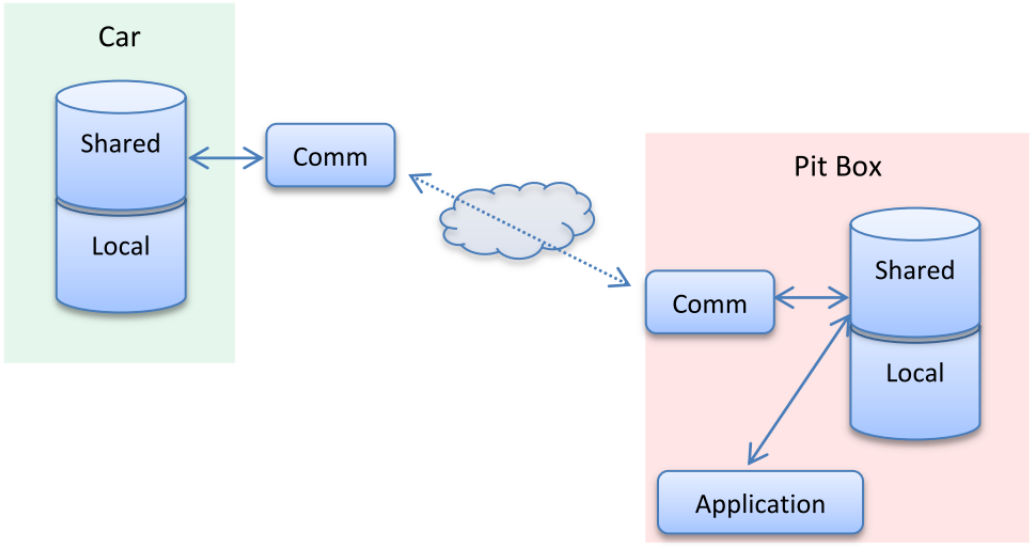


Figure 3.4: Pit Box and RtDB

Having access to the shared items of the RTDB, it can be used to display, over a pictorial representation of the track, perception information, like the vehicle position, the localization of the obstacles, the sign in the traffic panel, etc. Also we can have access, in real time, to the actuation orders sent by the high-level control subsystem to the low-level control subsystem. This information can be made available locally at the vehicle's laptop, but often is uncomfortable to visualize it there, specially if the vehicle is moving. This way, the pit box can be a very convenient way to follow the vehicle's perceptions and decisions.

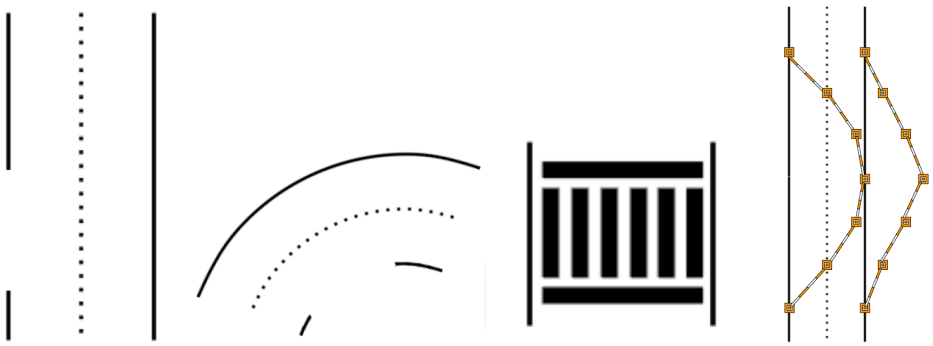
Navigation of the vehicle is based on roles and behaviors. The pit box can be used to choose the vehicle's role to be used in a given moment. For instance, it can be used to emergency stop the vehicle.

The full graphical pit box application has not been developed yet. However, it was implemented a simple command line application to test the *Pit Box* approach. It gives the ability to send orders to the car and some tests show that we can drive the car and set roles easily. Other information, like battery status, current internal states, like active behaviors and roles can also be viewed on this application.

3.4 Track Representation

The autonomous driving competition is disputed in a track that resembles a real world track (like the one described in section 1.2). To increase robot reasoning an internal representation of this track is required, e.g. to support trajectory planning. In general, a track is

considered to be composed of a set of road sections interconnected through their extremities [46]. The connection line between two adjacent sections is called a border. Sections can be partially overlapped. Thus, a border can be common to more than one pair of sections. The track model has the purpose of representing the properties of the sections, like distance, curvature and line type. Information about connections between sections is also stored. As proposed in [46], there are different types of track sections. The **Straight Section** and the **Curve Section** are the two more common ones. As their names suggest, they have a single geometry, a straight, the former, and a unique curvature, the latter. Both types of sections are characterized by three delimiting lines, defining the limits of the driving lanes. These delimitations can be solid, dashed, or a combination of both. An example of each type is depicted in figures 3.5a and 3.5b. The **Crosswalk Section**, as their names suggest, defines track section related to the crosswalk. The **Road Work or Maintenance Section** describes a piece of the track that is delimited by a set of pares of orange road cones. These cones are connected by red and white strips, where the first and the last pair are superimposed on the road line (Figure 3.5d). When the car enters in the working area for the first time, it does not have any information about it. Hence, do deal with these situations (in a temporarily way), the **Unknown Section** has also been implemented. This will also allows the future challenge of providing the robot with SLAM (Simultaneous Localization and Mapping).



(a) Straight (b) Curve (c) Crosswalk (d) Maintenace

Figure 3.5: Examples of support sections in this structure

The figure 3.6 shows the competition track sections division. This can be divided into two disjoint sets, sections (S), from *a* to *g* (green letters), and borders (B), from *1* to *6* (red numbers). Note that sections *f* and *d* are partially overlapped; the same for sections *a* and *c*. Figure 3.7 represents the same track, seen as a bipartite graph, where the two type of nodes represent the sections (rectangles) and the borders (circles). Actually, each border is represented by two nodes, corresponding to the two driving directions. Sections are only connected to borders and borders to sections, thus, there are no odd-length cycles in the graph.

The car is represented by a token in the system and is placed in a section. In the case of moving from section *a* to section *b*, the border is equivalent to pass from section *b* to section *a*, and so, in this case the borders *1e* and *1d* could be collapsed into *1*. Although, in the case of borders *3e* and *3d* between sections *a*, *c* and *g* the same collapse is not possible. If these borders were collapsed into *3*, the car can move between each pair of sections in the set [*a*, *c*, *g*]. The car is not allowed, however, to pass from the section *a* to *c* without pass through *g*,

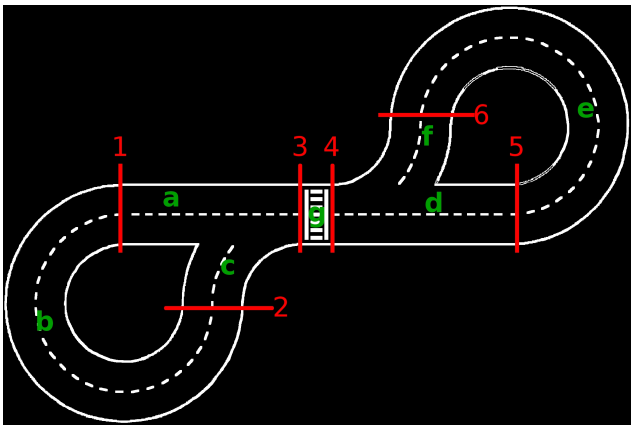


Figure 3.6: Portuguese Robotics Open track sections (green letters) and borders (red numbers)

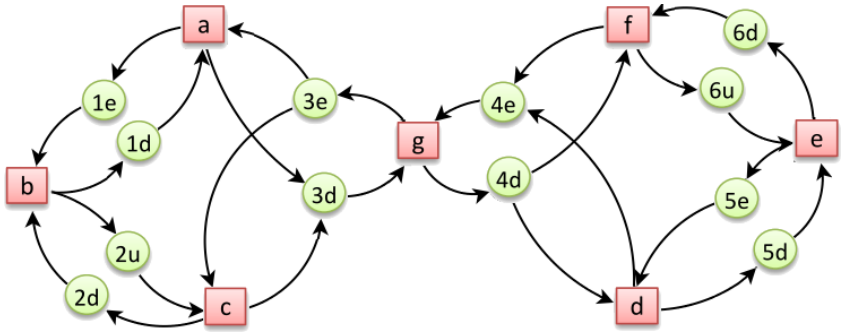


Figure 3.7: The Portuguese Robotics Open Track represented as a bipartite graph. The rectangles are the sections and the circles are the borders, connections between adjacent sections.

---

but it can pass from g to a or c. Therefore, the borders *3e* and *3d* are not equivalent, and so, the borders cannot be collapsed. The same is applied to borders *4e* and *4d*.

It must be noticed that the proposed bipartite graph is not a Petri net. If it was the case, then the sections and the borders should be, respectively, the places and the transitions of the Petri net. In a Petri Net, when a transition fires, a token is added to all the successor places and so, in figure 3.7, if the car crosses border *4d*, coming from section g, it goes simultaneously to sections f and d.

In [46] a Petri net was used to model the track, but the border was further divided than is proposed here. For instance, border *3* in figure 3.6 gives rise to 4 transitions, corresponding to the 4 possible section crossing. It is an equivalent but heavier representation.

When the car moves inside a section, its view is relative to the section. Thus, a coordinate system is associated to each section. The type of coordinates used depends on the section's type. For a straight or crosswalk section cartesian coordinates are used, while for a curve section polar coordinates are more appropriated.

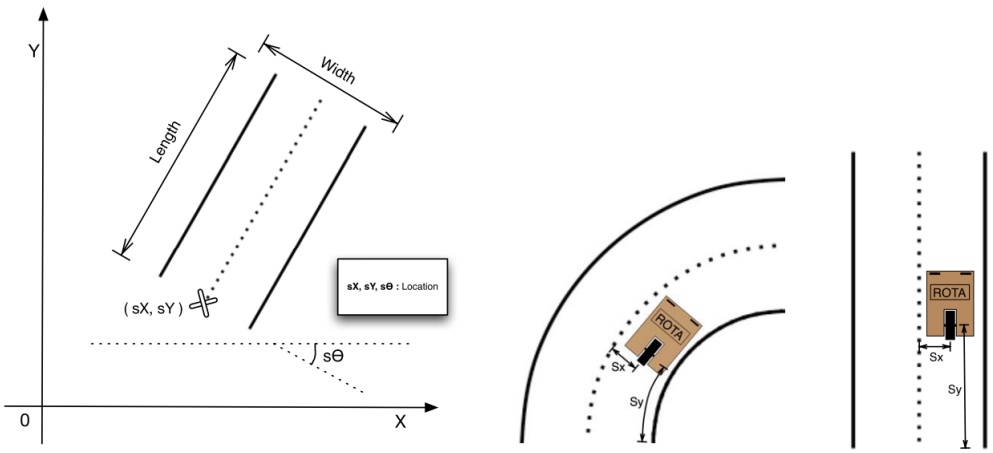
Each section is positioned in relation to a global coordinate system in order to provide information to the planning. The coordination system uses the center line, in one side of the section, as *Start Position* (figure 3.8), where:

Sx is the distance of the car to the center line.

In curve sections Sx is the radial distance between the center line of a curved section.

Sy is the distance of the car to the section start point.

In curve sections, Sy is the angle between the start of the section and the radial line that passes through the center of curvature and the given position.



(a) Section Properties (b) Car localization in sections

Figure 3.8: Coordinates Systems[46]

The car has also its own coordinate system. It is centered in the car's rear wheel, having the Y axis along the front axis of the car, as depicted in figure 3.9. Using the three coordinate systems make it easy to position the car globally or in a given section.

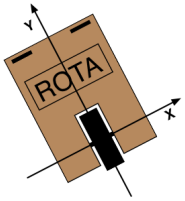


Figure 3.9: Car Coordination System [46]

To enhance the software flexibility, the track representation module was enriched with the possibility of saving and loading the track from a configuration file. The configuration file has been defined to support a list of sections and all the connections between them. A template of the configuration files is in appendix A.2.

3.5 Perception

In order to support its decision-making the car has to know the environment where it moves and locate itself on it. In the typical competition scenario, the track shape is known in

advance, but, there are elements that are only known in competition time. This is the case of the traffic signs, the obstacle location and the road work area location. The perception subsystem is responsible for, based on the available sensor data, locate the car in the track and map the dynamic elements.

The vision system is the main source of sensor data and thus, most of this section will be devoted to it. Additionally, odometric data from the traction wheel is available and used in the car self localization procedure. The car platform also has two infra-red distance sensors and two infra-red ground sensors but, currently, they are not being used.

The next chapter covers the vision system in detail and therefore, in this section, only a brief description will be given. The car has two cameras, one to look at the road and another to see the traffic signs. The cameras are equal and thus, the image acquisition process is the same. However, the type of processing applied to the two images are completely different. In the road image, *scanlines* are used to look for points of interest. At a higher level, these points are used to identified the lane limits. In the sign image, a template matching procedure, based on previously acquired sign images, is used to identified the traffic sign values.

The perception subsystem is structured in a graph of perception modules, also called perceptors, each one defined as represented in figure 3.2. The leave nodes of this graph use sensor data as the stimuli and produce some kind of perception data. The other graph nodes use perception data from other nodes or from a previous interaction of themselves to produce their own perception data.

An important perception module is the one responsible for the self localization of the car. It uses the points of interest given by the road vision pre-processing module and a lane model to extrapolate the car position in the lane and then using the world map try to correct the global position. To give a better position estimative a Kalman filter has been used.

3.6 Low-level communication handler

The low-level control subsystem has the aim to make the bridge between two subsystems, the high level control and the sensing/actuation as depicted in the figure 1.3. The sensors and actuators subsystem works in a CAN network containing several nodes, sensors, actuators and a USB gateway to communicate with the laptop [15].

The communication between the high level and low level is divided in two handlers. Each one is implemented in a thread. The Transmission handler (Tx handler) sends orders designated by the high level to the actuators. The Receive handler (Rx handler), pickup data from sensors and pass it to the high level. Both communications uses the RTDB as depicted in the figure 3.10. The RTDB items used in this communication are, *RtCarVelocity*, *Rt-*

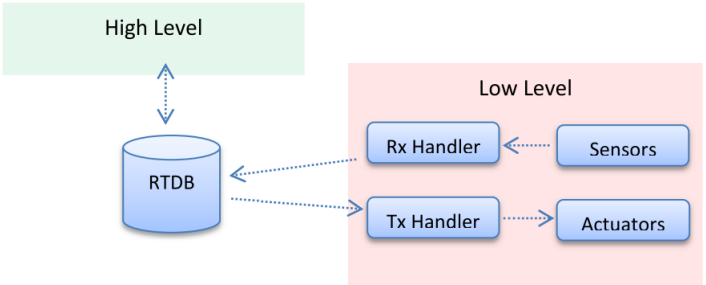


Figure 3.10: Low-Level communications

CarLights, *RtCarKinematicData*, *RtCarSensorsData*. The Tx Handler uses the former two. The *RtCarVelocity*, is used to send the velocity to the rear wheel and the curvature to the front wheels. The *RtCarLights* to turn on and off the car lights, on the platform. The latter two are used by Rx Handler. The *RtCarKinematicData* stores information about kinematic data, e.g. the total travel distance (odometric data). The *RtCarSensorsData* has information about the front and ground sensors.

3.7 Decision and Action

In figure 3.1, the decision and action block, usually referred as the *agent*, is the principal piece of intelligence on the system and is composed of several modules that control the actions of the robot.

At this stage of the processing cycle it is assumed that the perception block has already concluded its work and so the world state, as perceived by the vehicle, is updated. Based on it and on the goals, this block has to choose the best possible actuation orders and send them to the actuators.

The agent construction is done through the notions of *role* and *skill*, which can be seen as high level and low level tasks, respectively. A skill, also referred as a basic behavior, represents the vehicle's ability to perform a simple task, like *follow a delimiting line*, *follow a motion plan*, and *change the driving lane*.

Skills are combined to build a role, which can be seen as the vehicle's ability to accomplish a goal. For instance, for participating in the autonomous driving competition of the Portuguese Robotics Open, 3 roles were constructed, one for each stage of the competition. Roles are built as finite state machines, were the states correspond to skills and the transitions represent the events that cause a change in the skill that is being executed. Figure 3.11 represents the role used to perform the third stage of the above referred competition.

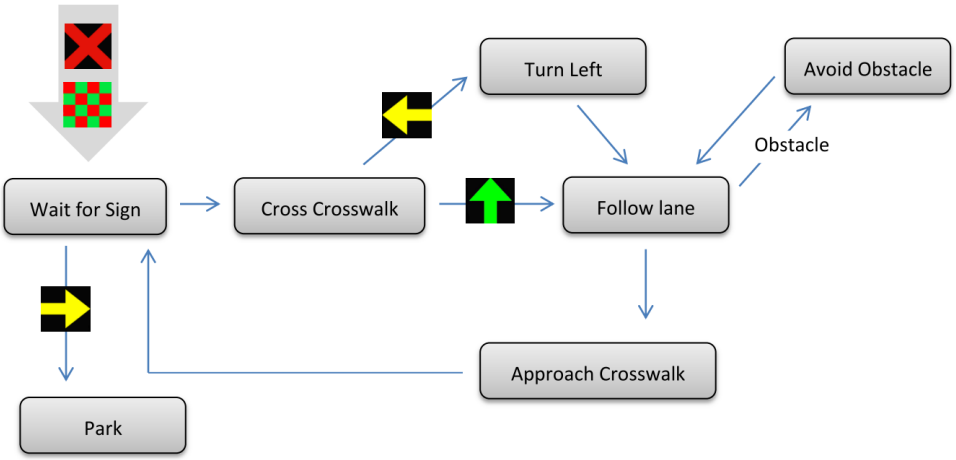


Figure 3.11: State Machine from Role used to run in the 3rd round of Portuguese Robotics Open (2010 edition)

Actually, a different agent process exists for each different role. Once the pitbox node is fully operational, a new level is planned. The idea is to allow for dynamically choose the role through the pitbox. A shared RTDB item will be used to allow the pitbox to force a role to

be executed by the vehicle. On the vehicle's side, the agent process look at the RTDB to 'see' what role to play.

Planner

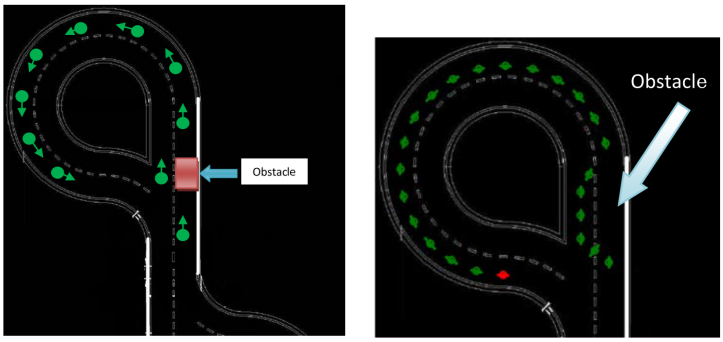
The movement control skills of the vehicle can be splitted into two categories. In one side, the steering control capabilities are directly driven by the distance of the vehicle to the delimiting lines. For instance, the vehicle can follow the right delimiting line, keeping a distance from it by a fixed value.

On the other hand, the vehicle's movement can follow a given motion plan. For instance, when the car turns left and as the lane has not delimiting lines, it uses a motion plan to perform the action. A motion plan is given by a sequence of points by which the car has to pass. Generically, each point is represented by its world coordinates and a posture of the car in that point.

To create a motion planner it is necessary a world map, in the case the track representation that is presented on the section 3.4. In [47] is presented a planner, which uses a track model similar to the proposed. This planner produces two peaces of information: a sequence of **wayPoints**, and between two consecutive, one **wayState**. The **WaiPoint** has the information about the world coordinates, through which the car should pass, and the correspondent heading at this point (usually to simplify the movement to the next WayPoint). For two consecutive wayStates the planner produce a **wayState**. The wayState gives information about the distance that the car must travel, and its curvature, in order to go from one wayPoint to the next one.

The figure 3.12, shows on the left, a motion plan result to perform the entire curve, (the wayPoints are presented). The distance between wayPoint can be configured in the referred planner. The planner also has the ability of producing new wayPoints between the existent ones. Hence, different levels of resolution of the motion plan can be created. The figure on the right shows the same motion plan, but with more resolution.

The car can now use the *follow a motion plan* behavior to drive through the sequence of wayPoints.



(a) First step WayPoints (b) Second step WayPoints

Figure 3.12: Half lap plan calculation [47]

A division of the planning in levels may lead to a more responsive system. For instance, if some event occurs, the world representation is updated and two things can happen: 1) the update is relative to a change close to the car position, (such as the detection of an obstacle)

and, in this case closest wayPoints has to be re-calculated; 2) the update is relative to a new goal (such as traffic sign change) and in that case the whole plan has to be recalculated.

3.8 Process Synchronization

The proposed architecture runs concurrently but the different processes are not completely independent. The time is discrete, all the processes run in cycle, and some processes use information produced by other processes. For instance, the agent uses data produced by the perception, which in turn uses data produced by the vision system. Thus, there are precedence relationships among the several processes and hence a mechanism to manage those relationships is required.

Another important issue in the process management is the CPU time consumed by each process. It must be guaranteed that the overall cycle processing time is lower that the available cycle time. The road image is the main source of sensor data to control the vehicle, and the road camera is programmed to grab 30 frames per second. So the camera is used to impose the cycle in the control software. Thus, the overall available cycle time is 33 milliseconds.

The Process Manager (PMan) module, also inherited from the CAMBADA project, is the responsible for the process management. This module is not explicitly represented in figure 3.1, since it is present in all the system, controlling CPU time of each module. Nowadays the use of hyper-threading and multi-core systems turns the use of simple solutions, like priorities, not sufficient to manage processes being now required explicit primitives for synchronization. The PMan is an abstraction layer to the system primitives to help the developers, and also add extensions to the native services. It is implemented to support the Linux system since it uses the Linux primitives and modify system priorities to best performance in a General Purpose Operating Systems (GPOS) [1] [48]. The PMan API makes available a friendly way to synchronize the processes. The initialization of PMan is performed using **PMAN_init()** and terminated with **PMAN_close()**. It is necessary to fulfill the PMan tables with the processes that will be synchronized and also with the precedences between them. The precedences are defined using **PMAN_procadd()** method. After the table initialization, each process has to be registered using **PMAN_attach()**, to insert its *Process identification (pid)* in the table. Likewise, there are methods to remove processes from the PMan table, **PMAN_procdel()**, or only its pid, **PMAN_detach()**. After processing completion, each process must inform the PMan manager, using **PMAN_epilog()** or in case of the master process, **PMAN_tick()**. The master process is an high priority process in the system, which is put in execution whenever processing data is available.

To increase the software flexibility, was defined a configuration file for the process manager and a *PManFactory class*[2]. The configuration file contains necessary information to create the PMan table (template on appendix A.3). This information is only composed by the name of the processes and its precedences. The *pid* is not included in the configuration file, because this is only available during running time. The *PMan factory class* is a C++ Factory, created to encapsulate the use of configuration files (figure 3.13). It reads the configuration file and creates the necessary structure to the work of PMan. When called, the Factory is also used to provide the pid of the process to the PMan manager.

[1] The GPOS are time-shred processors (nowadays for multi-core ones) that deal with lots of things and "optimized to manage heterogeneous classes of resources (CPU, disk, network interface, etc)"

[2] The Factory class implements a design pattern, [49]

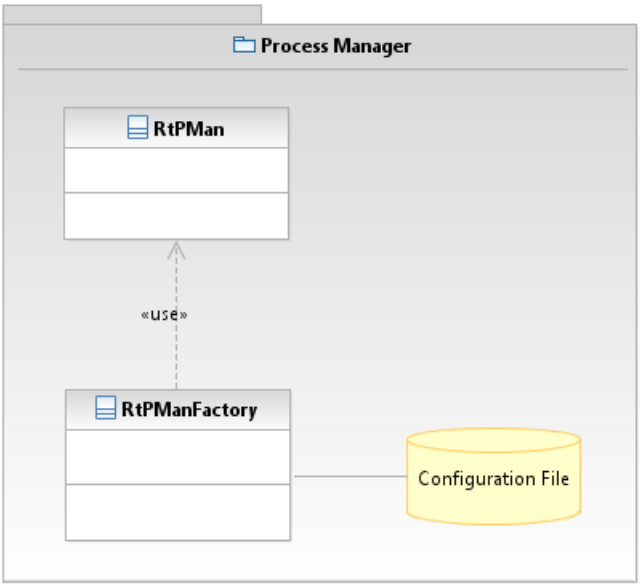


Figure 3.13: PMAN Factory class diagram

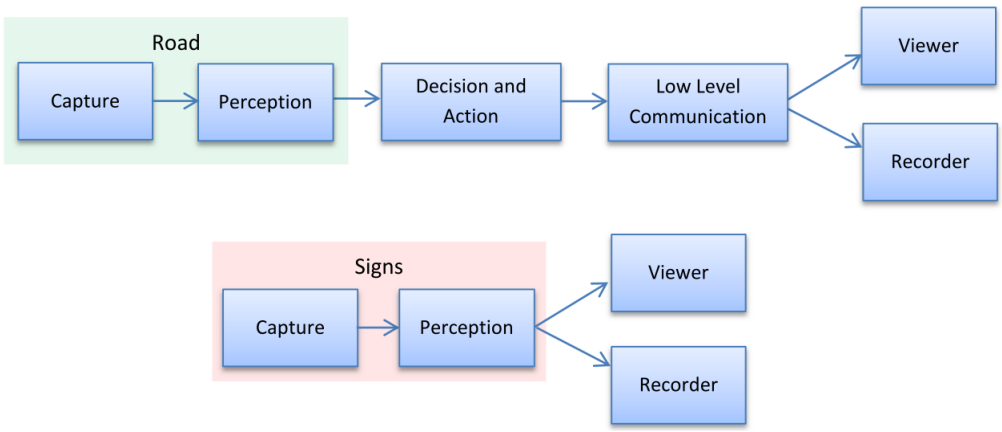


Figure 3.14: Rota actual Software cycle / sequence

As stated in the beginning of this section, there are some dependencies in the modules. The capture process connected to the road camera is the master process, which defines the software cycle. However, the signs camera is independent and has to work asynchronously. The frame rate of the cameras are different, the road camera works with 30fps and the signs camera with 15fps. The figure 3.14 depicts the dependency graph between the defined modules.

The main source of the information to drive the car comes from the road images. Therefore, the capture module is the master process, the availability of a new frame trigger a new system cycle. This starts with capture, which performs image pre-processing. Then, takes place the perception, which extracts information from the pre-processed image to be used by the next module, decision and action. This creates orders to the car that are sent by low level communication handlers to the base. At the end of the graph are the viewer and recorder,

which can be executed in parallel.

For the signs processing, the capture also precedes sign perception, and the latter precedes viewer and recorder. There is no explicit dependency between the road and the sign, although, sometimes, the decision and action module need information not only from the road but also from the signs to make a decision, e.g. in the crosswalk. In this case, the decision and action module has to wait for the sign processing completion (not a blocking wait). It can continues driving, ignoring the need for signs on this cycle, hopping that in the next cycle the information is available, or in the worst case, stop the car and wait for the sign processing completion.

Chapter 4

Vision system

Human vision is a very complex system, which together with other senses provides information about the surrounding environment. The human eye has a fixed field of view, and normal behavior of the human visual perception is to focus on several elements on environment, track them and analyzing them continuously. The remaining areas, the peripheral vision, have small accuracy, but the brain is attentive to sudden movements or large luminosity variations on this areas. Thus when we want to look something specifically, our brain focuses these elements, so, this action needs some part of our reaction time. Hence, there are two important aspects. The human vision does not process all the field of view on the same way. The human eyes are adapted, by focus on what is important to the current task. And so, "Which aspects of the rich visual stimulus should be considered to help the agent make good action choices, and which aspects should be ignored?" [1]

Most of contemporary robots are dependent on vision, which is provided by cameras that try to reproduce the human eyes. The cameras have lens to enlarge or compress the field of view, and respectively decrease or increase the definition of the image. Other cameras capture large images, using a high density matrix to improve definition, in which cases the image result have a large dimension. A large image takes more time to transfer, and so, such cameras usually have techniques to transfer only parts of image and reduce the communication time. Otherwise, the delay to start the image analysis could be very high; such delay could turn the image obsolete.

To control the robot, the main data source comes from images, captured by the cameras. It is then necessary to extract information from these images to feed the robot reasoning. This chapter focuses on the Vision system, briefly presented in the chapter 3. The vision system includes the capture, perception and part of viewer and logger presented on figure 3.1.

4.1 Vision System Architecture

Figure 4.1 depicts a general overview of the architecture proposed for the vision system. The general idea of the architecture is to separate the Vision System in three main module improving, this way, modularity, extensibility and reusability and easing the development process. The three modules are the image capturing, the image processing and the image visualization and/or recording.

The architecture unfolds into three processes, one for each module referred above, playing around a central set of shared image buffers. The number and type of the buffers depend on

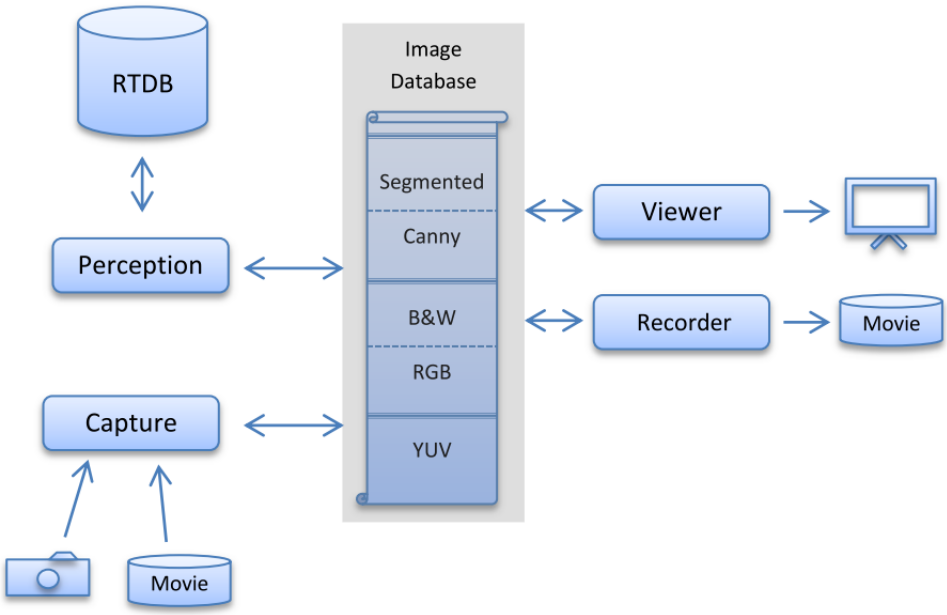


Figure 4.1: Shared Image architecture

the processes. For instance, if the image is captured in RGB format and the processing is done on black and white, at least two image buffers are needed, one for each type.

As described in chapter 3.1, the ROTA vehicle has two cameras, one to capture road images and another to capture traffic sign images. The type of processing applied to the two images is quite different, while the capturing and the visualization/recording procedures are the same. The proposed architecture makes it easy to set up the system, since only four pieces of software are required: one for the capturing, one for the visualization/recording and one for each processing.

Depending on the purpose, not all the processes need to be launched. A remote control application in conjunction with the capturing and the recording processes can be used to make a movie of a run. In this case, the image processing process is not required and could be not launched. During competition, the visualization is not useful and just consumes CPU time. However, at startup, it is useful to have an image to verify if things are right. So, we can launch the three processes at startup and then kill visualization.

4.2 Images Database

As referred before, the number and type of image buffers available in the shared memory depend on the processes that are using them. To support this versatility, aside with the image buffers, the memory contains a configuration header. This header establishes the number of frames and the properties of each one. One of the images is referred as the *original* and corresponds to the one acquired by the capturing process. The other images are derivative and so produced from the original or another derivative. The capturing process, described in section 4.3, is responsible for all the required image conversions.

The image database was defined to store any amount of images, as depicted in the figure 4.2. The global parameters give information about the number of available images slots, and

the base size. This base size represents the size reference for all the other images, i.e. all other images are held on a scale of this. Next, we have a list of image headers (one per slot/image). This local header contains the name, a scale factor, number of channels, flags and a stamp. The name of the image also represents the image format. The number of channels is used to define if it is a black and white or colored image. A scale factor is a real number that multiplied by the base size gives the current image size. The choice of scale number, instead of image size, is due to the use of mapping from metric to pixel and pixel to metric. If the image was distorted, this mapping would also be distorted, so a big overhead would be introduced. A flag is provided to mark the image as active or not, which is useful to prevent unnecessary processing of unused images. The modules can use the Stamp to control if any image is lost, and also should be used to synchronize the debug information in the log files. After the headers comes a list of image buffers for the image data.

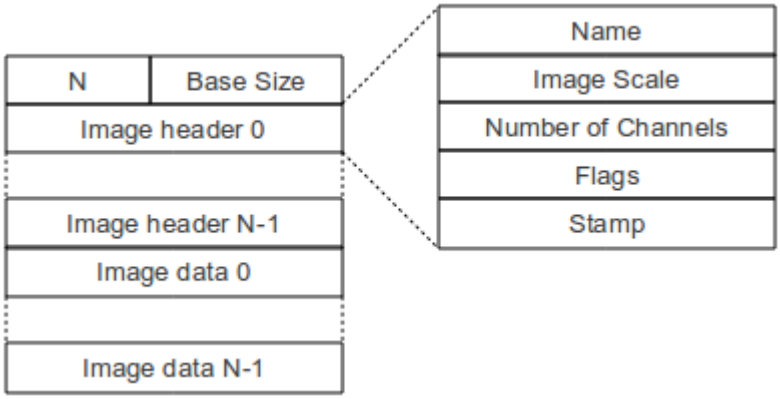


Figure 4.2: Shared Image Implementation

After the software startup, all this information (with exception of image data) is static. However, there is a configuration file that stores all these settings, the required frames and its properties. An example of a configuration file is given in appendix A.4.

A simple API was developed. To create and initialize the image database, is used the method **RtImageDatabase()**, it reads the configuration file, and sets up all the internal structures. Using the image name, the method **GetImage()** gives access to the image data. The method **GetOriginalImageName()** returns the name of the original image. This API also provides two methods to get two distinct lists, **ListConvertImageNames()**, **ListOtherImageNames()**. The former returns the list of images automatically convertible (defined on the configuration file). The latter, gives a list with the remaining images (except the original one). The prototypes of the Image Database related methods are:

RtImageDatabase(string configFile);

SharedImage* GetImage(string imageName);

vector <string> ListOtherImageNames();

vector <string> ListConvertImageNames();

string GetOriginalImageName();

Comparing to the old cameras software, the image database does not introduce any relevant delay to the software. A test was performed involving all the software, the robot control, capture, processing and visualization of the images, and at any time the software cycle time was exceeded. This improves Modularity, Flexibility and Expandability, since the images are stored in shared memory, being accessible by all the software. A direct advantage of this, is the construction of a new layer in this hierarchy without any change in lower layers, and the expandability is only restricted by computational resources. It also allows modules exchange without modifications on the adjacent ones, improving flexibility in the development environment.

4.3 Capture Process

The capture process is responsible for the frame acquisition and frame conversions. At every cycle a frame is acquired from a camera or a video file and stored in the frame memory. The image in this frame corresponds to the one referred as the original. Then, all the derivative images are produced, and this ends the capture cycle.

If the original image is acquired from a camera, the cycle time is determined by the frame rate set on the camera. If it is acquired from a video file, the cycle time is defined by a timer.

Figure 4.3 depicts the class diagram that supports capturing. The new version structure is now more extensible, and modular. There is an abstract data type, **RtCapture** that, when extended, provides access to a specific type of images source. There were three types

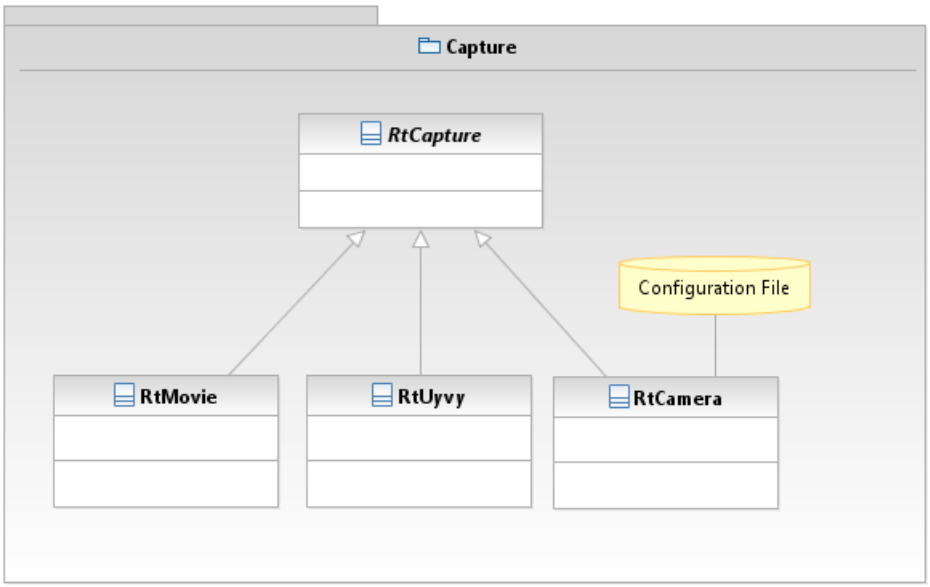


Figure 4.3: Capture Class Diagram

implemented: **RtCamera**, **RtMovie**, and **RtUyvy**. **RtCamera** is used to access camera through library libdc1394. This includes cameras that use the IEEE 1394 interface [1], but also some USB cameras that use libraries libdc1394 and libusb together [2]. **RtMovie** make use of

[1] The interface is also known by the name FireWire (Apple)

[2] The USB cameras has to support the IIDC 1394-based Digital Camera Specification.

the system codecs through the OpenCV library, and so, can read the majority of the movie files formats. Finally, **RtUyvy** was implemented to allow the use of raw movies acquired in the past. This is a homemade movie format composed by sequence of simple raw images, stored in the YUV422 format.

The capture process has to grab the image and make the conversions established in the header of the image buffers. As stated before, the required conversions are defined by a configuration file and thus are static after startup. On the other hand, the same buffer can be used by more than one consumer. So, the conversions are applied to the entier image by the capture process, just after the grabbing. The figure 4.4 shows the work flow of the Capture process. The first step is grab an image, using the method **ReadFrame()**. Then, makes only the conversions between images that are active, since an inactive image is not used by any module. Finally, all the pre-processed ones, in the image database.

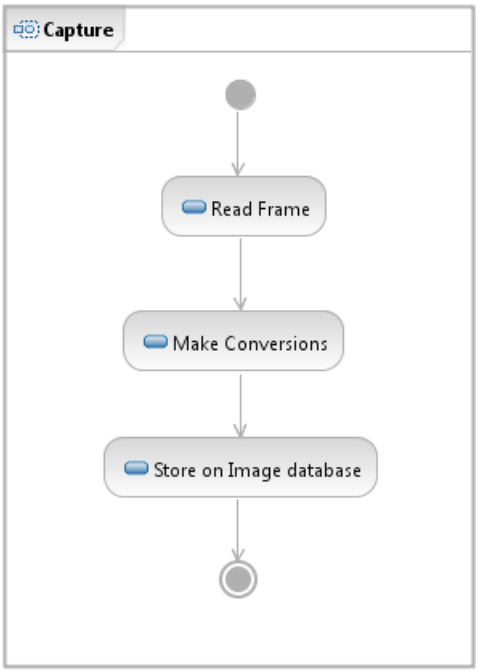


Figure 4.4: Capture work flow

The conversion functions are defined in a matrix, as depicted in figure 4.5. This matrix is a mapping between two images, identified by their names, and a function that makes a conversion between these formats. Instead of using the functions from the old software, the functions from the OpenCV library [3][50] are now used. These are optimized to the image processing, and so, they are faster.

In order to make the module even more flexible, save and load configurations from camera are implemented. A configuration file template can be found in appendix A.5.

[3] OpenCV (Open Source Computer Vision)

	RGB	YUV
YUV	RGB_to_YUV()	
RGB		YUV_to_RGB()
B&W	RGB_to_B&W()	YUV_to_B&W()

Figure 4.5: Matrix of color conversions

4.4 Visualization and recording

As mentioned before, the vision system was divided into three modules. Throughout this section we will describe the visualization and recording processes.

The visualization process has only a simple task, to show the images stored in the Image database. The figure 4.6 shows a simple GUI application, the viewer, used to show the images to user/developer. This is a very basic application, the only available control is to force an image to be processed or not (active or inactive). As referred in a previous section, the active flag is used to tell the producers whether it should or not produce the image, and consequently to the consumers if they should or not ignore this image. In the startup, the viewer shows all the images. In the case presented in the figure, four images are in the database: YUV444 (the original), RGB, Black and white and finally the *Debug* (used to print some debugging). The flexibility introduced in the architecture of the vision system, allows this application to be opened and closed, at any time, without having to restart any other process.

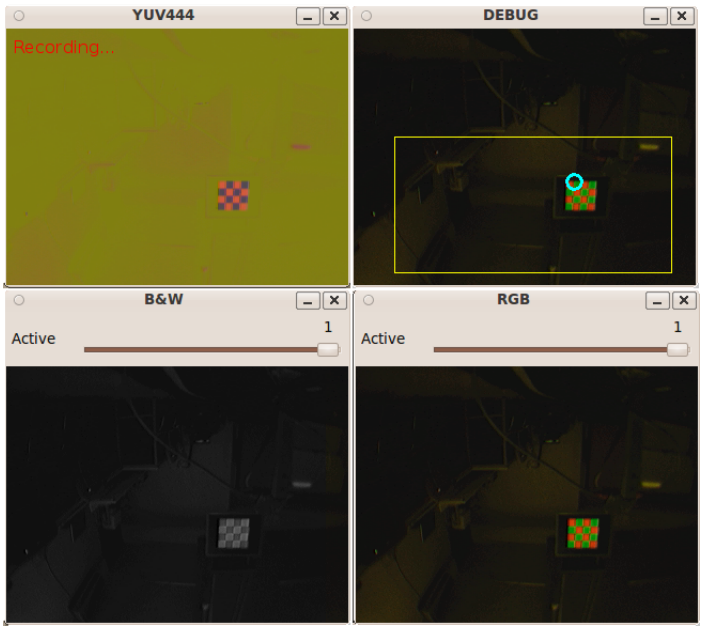


Figure 4.6: Viewer, a simple GUI application to show images

The recorder module, which contains the encoder algorithms, was reorganized as the capture module to improve its extensibility and modularity. The class diagram of the Recorder is depicted in the figure 4.7. There is an abstract class **RtRecorder** that, when extended, gives the actual recording formats. There are two classes implemented: **RtRecorderMovie**

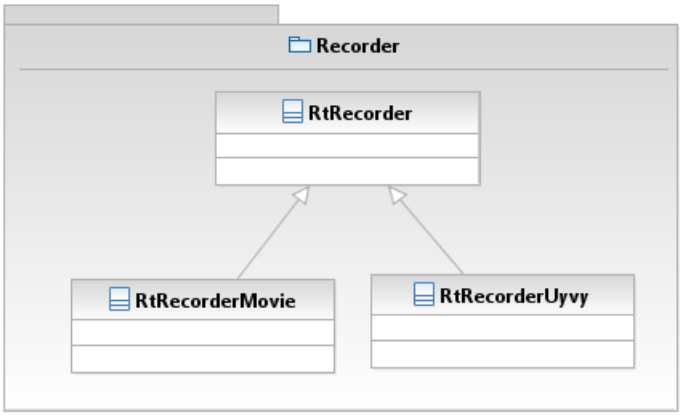


Figure 4.7: Recorder Class Diagram

and **RtRecorderUyvy**. The **RtRecorderMovie** uses the system libraries to encode into the available formats. The **RtRecorderUyvy** format, implements the *uyvy* encoder, that was added to support the oldest video format.

There is no special GUI application to the recorder purpose, and also making one video with different image sources and/or formats is not a requisite. So, the recorder function is integrated in the viewer application, in order to simplify its use. When we start the viewer, we can pass an option to select the source image (using its name) to be used as video frame. At running, there is a *hotkey* to pause and continue the recording process. If the recording is active, a simple information message is displayed in the top left corner of the original image, as in figure 4.6. However, once the pitbox is developed, the recording process can run in background and be controlled remotely through it.

The viewer and the recorder, are also controlled by the PMan. The intent of both applications is to present or save information, usually processed. This information is only available at the end of the system cycle. Both applications are also not a priority to the robot, so this only must run with the remaining time of CPU. The viewer and recording processes could run concurrently in the system as depicted in the figure 3.14.

4.5 Road Image Processing

The majority of information used to drive the car comes from road perception. The processing is splitted into two different layers. In the lowest layer, the road image is searched for Point of Interests (POIs), based on a set of scanlines. The detected POIs are stored into the RTDB. The second layer is actually a module of the perception process and uses the POIs as input data.

4.5.1 Points of interest extraction

Often, it makes little sense to process the entire image. In one hand, in order to have a tight control of the vehicle, a short cycle time is required and processing the entire image can be very time consuming. On the other hand, in general, only parts of the image are relevant. The way used to process the image is through *scanlines*, defined within a Region of Interest

(ROI) in the image.

The general idea is to have a kind of intelligent perception, such that the ROIs are not statically defined, but can be adjusted by the perception system. Each image perception element follows the schema depicted in figure 3.2. Its activation condition is the availability of a new image in the image buffer. The stimuli are the image and a ROI defined in the track plan. Finally, the output is a set of POIs, which are stored in the RTDB.

The ROI is defined in world coordinates, but due to the camera distortion, this region cannot be reproduced in the original image. In the figure 4.8 are depicted, on the left, the original image in the track plan, and on the right, the image on world plan (undistorted). The ROI in the world coordinates is a rectangle, but when converted to the pixel coordinates, this is transformed in a distorted trapezoid. As can be noted on this figure, the red and green areas are similar in both images. In order to compute the entire area defined on the requested area (on track plan), the image processing defines a new ROI that contains it. In the example the ROIs used are defined by the purple rectangle.

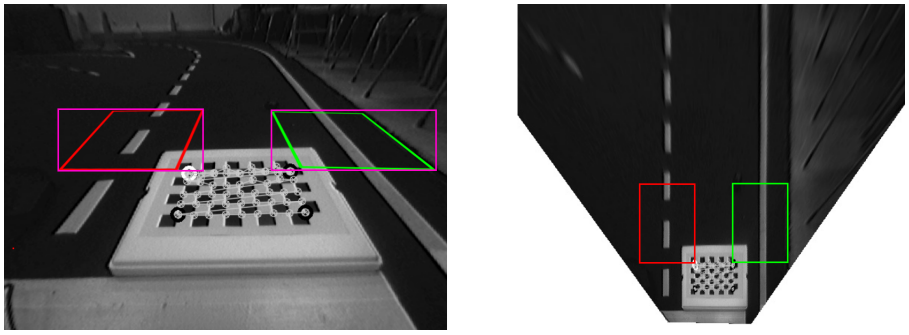


Figure 4.8: ROIs, real coordinates vs the pixel coordinates

The output POIs are given in coordinates of the real world. Since the *scanlines* are defined and applied to the image, a conversion from pixel coordinates to world coordinates is required. To accomplish the conversions two maps are required, one, the inverse mapping, to convert pixels to world in the track plan and another, the direct mapping, for the opposite conversion. A quasi-automated algorithm used to construct the two mappings was developed in another ROTA's project [51]. This algorithm firstly constructs the inverse mapping using a chess board. The result is a conversion matrix that maps each pixel to a point on the world (track plan). The direct mapping is obtained through an approximation algorithm using the inverse one.

The *scanlines* analysis is never applied directly to the source image. There are two different approaches available. One uses a gray-scale image and an edge detector algorithm to firstly produce a binary image representing the edges before process the *scanlines*. Presently, the Canny edge detector algorithm is being used. Since an edge detector algorithm is time consuming it is only applied to the ROI outlining the *scanlines*. Then, on the binary image, the *scanlines* are used to find the points of interest, which afterwards are converted to real coordinates and stored in the RTDB. The figure 4.9 gives an example of the process, used to look for the delimiting lines of the road lanes. In this example, there is only one large ROI, and only the middle line is being researched. The *scanlines* are purple colored and search from the left to the right. The blue colored dots denote where the first white pixels have been found.

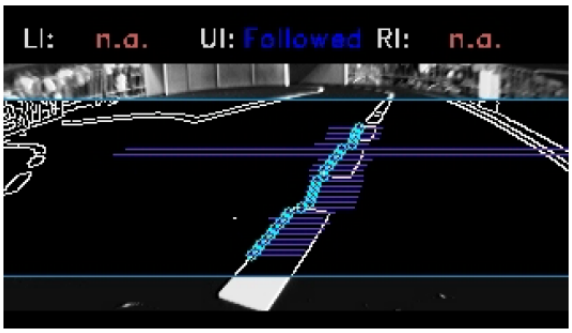


Figure 4.9: Image sensors

The other approach starts by making a color segmentation of the image. Then, the segmented image is analyzed using the *scanlines*, looking for relevant color transitions. For instance, color transitions from black to white can be used to identify the delimiting lines. As another example, color transitions from any color to green can be used to identify the obstacle.

The Work flow diagram 4.10 gives an overview of the sequence of operations performed by the process. As referred above, it starts with the activation condition, verified when a new image is available. Firstly, the process has to convert the ROI from the world coordinates to the pixel coordinates, and also transforms it into a rectangular ROI. Then, depending on the situation, edge detection or segmentation is applied to the region. Afterwards, the *scanlines* are used to search the POIs. *Scanlines* could search for pixels with one color or color

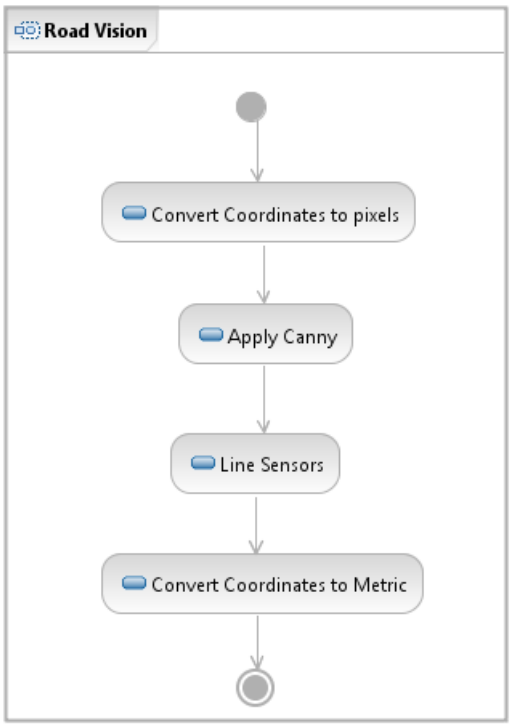


Figure 4.10: Road Vision work flow

transitions.The process terminates with the conversion of POIs found, from pixel coordinates to the world coordinates. This work flow is applied to each region from the requests.

4.5.2 Road Analysis

Points of interest, obtained through the method described before, are used for three different purposes: lane identification, obstacle detection, and cross walk detection.

Most of the time, the road has three visible delimiting lines, defining the two driving lanes. A road perception module is used to map the road in front of the vehicle, and so to inversely locate the vehicle within the road. For this it uses three ROIs, one for each delimiting line. Horizontal *scanlines* are used within the ROIs, going from left to right or right to left depending on the line. The obtained POIs, hypothetically points in the lane boundaries, are validated against a road model and a road profile is computed [52]. Note that, due to the absence of longitudinal landmarks along most of the track, only the transversal position, and horizontal orientation is computed. Figure 4.11 shows, both processes, POIs extraction and road analysis. On the left image is the image processed with the POIs superimposed, purple colored. At right, the POIs were converted to the metric system and marked with red dots. The Road analysis determine some properties from the select POIs, like the lane profile. The blue lines, depicts the lane extrapolation of each lane.

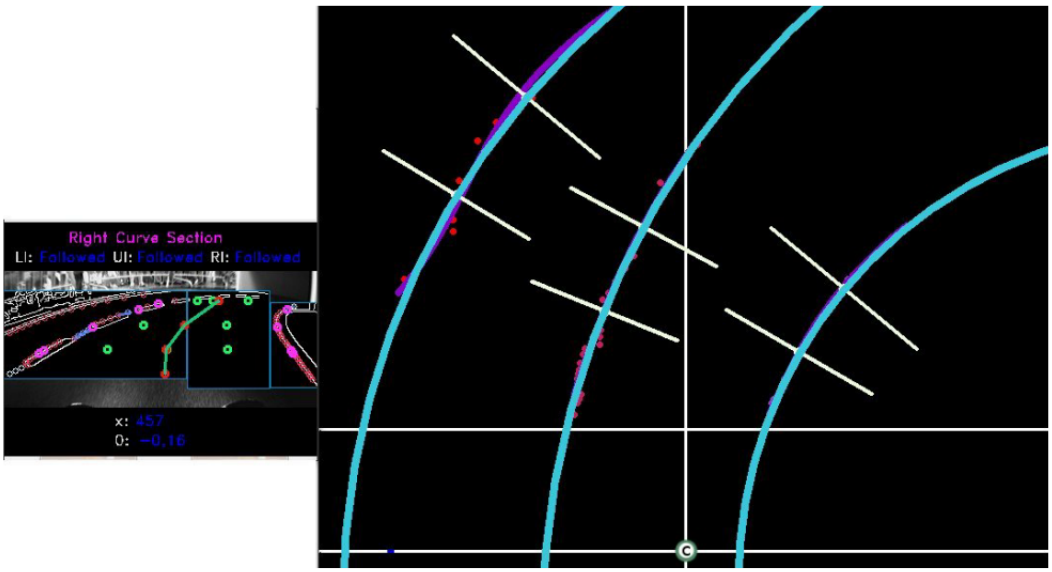


Figure 4.11: On the left we have the Road vision and the POIs with purple color. On the right the result of the Road Analysis, the red points are the POIs used and the blue lines the lane profiles [52]

There are also obstacles and the crosswalk inside lanes. The search for this is very similar to the lines. The main difference is that the last make use of horizontal *scanlines*, while this makes use of the vertical ones. The vertical *scanlines* can be used in both directions, bottom up and top down. On the left of figure 4.11, the green line represents the obstacle *scanline*. The same behavior is done to search for the crosswalk.

4.6 Traffic Sign Detection

The ROTA vehicle has a second camera used for detecting the traffic signs displayed both in a TFT monitor and in vertical signs put near the road on the outside. The camera is positioned in the rear of the vehicle, facing up, as explained in section 1.3. In the current version of the software only the TFT signs are evaluated, but the same algorithm can be used for the others.

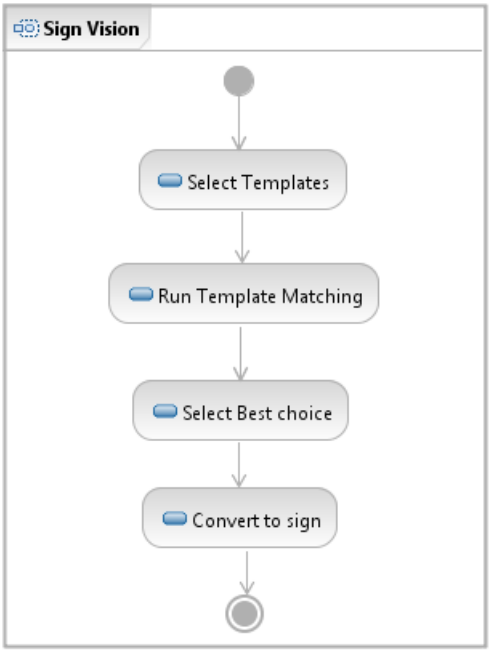


Figure 4.12: Signs Vision (Template Matching) work flow

Sign identification is obtained using a template matching algorithm. During the setting up for the competition and using the real track, images of the different signs, taken from different vehicle positions, are collected and become the templates used in the matching algorithm. Then, during the run, the templates are matched against the sign image and a voting algorithm is used to elect the winner. A minimum number of votes are required in order to validate the election. Additionally, a number of frames in succession are evaluated before a sign value is decided.

The OpenCV library provides three different methods to accomplish the template matching.

The **Square Difference Matching** method, given by equation 1, is based, as the name suggest, on squared differences between the template and the image.

$$R_{sq\_diff}(x,y) = \sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2 \tag{1}$$

In the equation, I, T and R are respectively the input image, the template and the result. Note that the result can be seen as a gray-scale image where a pixel close to black represents a point of good match. Thus, a perfect match gives a result of black (zero).

If the template and the original image were captured with different luminosity, the error on the result increases. In order to avoid this issue, $R(x, y)$ is normalized using equations 2 and 3. Normalizing the result reduces the effects of lighting differences between the template and the image, and the result is interpreted in the same way, i.e. the best match is the one close to value 0.

$$Z(x, y) = \sqrt{\sum_{x',y'} T(x', y')^2 \,.\, \sum_{x',y'} I(x + x', y + x')^2} \tag{2}$$

$$R_{normed}(x, y) = \frac{R(x, y)}{Z(x, y)} \tag{3}$$

The **Cross-Correlation matching** method, given by equation 4, calculates the cross correlation between the image and the template. Like on the previous method the cross-correlation result can be represented in a gray-scale image. On this case, the perfect match is the one that gets the more white color, and consequently, the greatest value represents the best match. The equations 2 and 3 are also used to calculate the normalization of this method.

$$R_{ccorr}(x, y) = \sum_{x',y'} (T(x', y') * I(x + x', y + y')) \tag{4}$$

The **correlation coefficient matching** method, given by equations 5 to 7, matches a template relative to its mean against the image relative to its mean. So, a perfect match will be 1 and a perfect mismatch will be –1. A value of 0 simply means that there is no correlation (random alignments). The figure 4.13 depicts a example of this algorithm.

$$R_{ccoeff}(x, y) = \sum_{x',y'} (T'(x', y') \,.\, I'(x + x', y + y')) \tag{5}$$

Where:

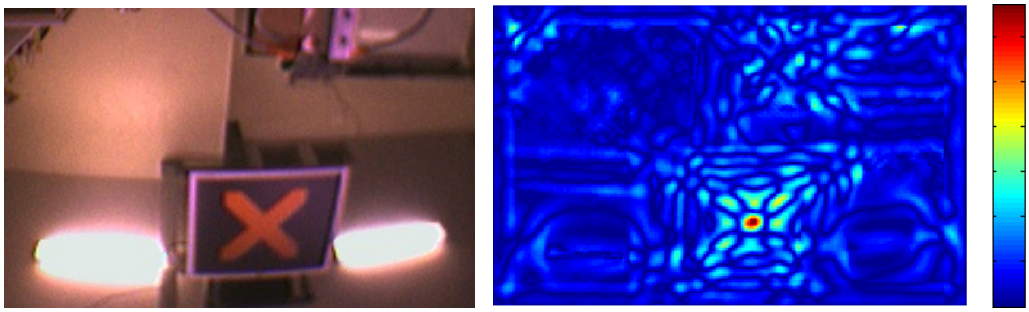
$$T'(x', y') = T(x', y') - \frac{1}{(w \,.\, h)} \cdot \sum_{x'',y''} T(x'', y'') \tag{6}$$

$$I'(x + x', y + y') = I(x + x', y + y') - \frac{1}{(w \,.\, h)} \cdot \sum_{x',y'} I(x + x'', y + y'') \tag{7}$$

The normalization of correlation coefficient also uses T' and I'. So, $Z(x, y)$ on equation 3, is given by equation 8.

$$Z(x, y) = \sqrt{\sum_{x',y'} T'(x', y')^2 \,.\, \sum_{x',y'} I'(x + x', y + x')^2} \tag{8}$$

The time required to apply a template matching algorithm, depends on both the number of templates and the size of the search image and this processing time can invalidate its application. Thus, the used approach tries to reduce the effects of both dependencies.



(a) Original image (b) Result of T.M. (colored map)

Figure 4.13: Template match result, the template used is the red sign image in figure 1.1b. Dark blue color represents the minimum and the dark red the maximum values. The best match are given by the maximum

---

The matching algorithm is not invariant to scale. This means that different templates for the same sign value should exist, taken from different distances and orientations between the vehicle and the sign panel. Four cases are considered: the vehicle close to the crosswalk coming in front; the vehicle around one meter away from the crosswalk when coming in front; the vehicle close to the crosswalk when coming from the right side; the vehicle around one meter away from the crosswalk when coming from the right side. Since there are 6 different signs, this could represent 24 templates. However, using the position of the vehicle and the phase of the run, the number of signs to evaluate at a given moment can be reduced to 3 or 6.

Based on the position of the vehicle, there is also an idea of the region of the sign image where the sign should appear. Thus, instead of search the total image, only a region is used to apply the template matching.

A configuration file is used to establish the templates to use, the image regions that should be search for a match and the default method to be used. An example of this configuration file is given in appendix A.5.

Chapter 5

Results

This chapter presents the results obtained from the implementation of the proposed architecture. It presents an overview of the implemented modules and applications, as well as the results of some tests done with the new architecture.

5.1 Scanlines

Since the very beginning the ROTA's software uses *scanlines* applied to the road image in order to localize the lane delimiting lines. In the version of the software developed under this thesis work, the novelty is the approach followed to accomplish that. Instead of using a fixed or variable ROI defined in image coordinates, a ROI dynamically defined on the real track is used to establish the scanning area. This world coordinates ROI is transformed to a road image ROI, which is then used to define the *scanlines*. The obtained list of points of interest (POIs) is finally converted to world coordinates.

An advantage of this approach over the previous one is that changing the camera or changing its location on the vehicle platform only implies a change in the direct and inverse maps used to coordinate conversions between image pixels and real world points. Moreover, rebuilding the maps is not a big deal, since an almost automatic procedure to do it exists. The map is used as a lookup table (LUT), so the time for the coordinate's conversion process is neglectable.

The set of existing behaviors to drive using the delimiting lines were rebuilt in order to follow the new approach. They were used during the autonomous driving competition of the 2010 Portuguese Robotics Open with good results - the vehicle never got out of lane when was driven by these behaviors.

5.2 Vision System

A GUI application was developed, to test and evaluate the implemented template matching algorithm. It follows the concurrent architecture proposed for the vision system (see figure 4.1). Figure 5.1 gives a snapshot of the application. On the left, one can see the sign (input) image annotated with relevant information. The yellow rectangle delimits the area where the template matching algorithm is applied. The blue rectangle represents the area where the algorithm should found the sign. The light blue circle denotes the area where the match has been found (it gives the center of the area where the best match occurs). The application also

displays, over the image, some informations, namely the expected sign, the signal found, the number of errors, the average time, the accuracy and the test progress. In the same figure, the window on the right side, composed of *track bars*, is used to control the application. The different controls allow the user to test every feature of signs detection software. The *method* selects the template match method to be used. The *threshold* bar allows the user to set the accepting threshold value. Depending on the algorithm chosen, the result has to be above or below that value in order to be considered a valid match. The user can also indicate the *expected sign*. To make several tests without restart the application, the *Reset* can be used to restart all the values that are used to calculate the results. The user can also select the number of samples used on tests, using the *test runs* bar.

A semi automated-test could be used to test the accuracy and time average of sign detection. Firstly, we define the two areas, yellow and blue, referred above. Then, select the method, threshold, expected sign and the number of runs to be performed. Now, use the reset slider to reset all the values and then the test is started. When the progress reaches 100% the test stops and the values do not change until the "reset" is used.

This application also supports the creation of the configuration file to be used by the traffic signs detection. The yellow area gives the valid ROI and the default method is given by the method selected in the control *method*. It also creates the template image, store in the configuration folder and associates the valid ROI and default method information in the configuration file.

(a) Test Information (b) Test Control

Figure 5.1: The Signs test application

5.3 Template matching

Considering the top speed an autonomous vehicle can reach — more than 2m/s in the case of ROTA —, a tight control of the vehicle steering is required. In the ROTA's vehicle a 33ms cycle time is used, controlled by the road camera. In such a small cycle time the use of a template matching algorithm can be too time consuming.

Thus, in order to evaluate the effectiveness and efficiency of template matching in the sign identification, two different experiments were done. One, had the purpose of evaluating the real cost of applying the template matching algorithm and also of evaluating the cost differences among the different template matching methods provided by the OpenCV library.

The other, was done to evaluate the efficiency of each method in the possible scenarios that the car may have during competition while facing the sign panel. Normally, the ROTA's software use images with dimensions 320x240 for both the road and the signs. However the idea of using the VGA format (640x320) was under consideration. Thus, it was also decided to measure the performance of the matching algorithms for both image formats. The experiments were conducted using the TFT panel and the signs that can be displayed on it.

For the first experiment 20 different templates were created, one for each sign value, for each image format, and for two distances between the vehicle and the TFT. The template dimensions for the different cases are summarized in table 5.1. Note that for the same distance and the same input image format the template dimensions are roughly the same, except for the chess sign. Actually there are 2 chess signs, which flash alternately. Both are a 4x4 chess board composed of green and red squares. The difference is a swapping between green and red. In the sign identification software, the same template is used for both, a 3x3 green and red chess board, and thus, the smaller dimensions of the corresponding template.

distance	sign	320x240	640x480
Small	STOP	30x29	39x37
	FRONT	31x26	39x37
	LEFT	33x29	39x37
	RIGHT	25x31	39x37
	CHESS	21x21	39x37
Large	STOP	37x30	91x70
	FRONT	45x34	91x70
	LEFT	40x30	91x70
	RIGHT	33x37	91x70
	CHESS	36x31	91x70

Table 5.1: Templates dimensions, the sizes are in pixels

Performance evaluation was done in terms of computation time. The results are summarized in table 5.2. The time depicted represents the average time required to process the largest template. In both cases a quarter of the image is processed. For the 320x240 image size, the area processed is 160x120 pixels and the template size is 45x34 pixels. With the VGA image, 640x480, the area processed is 320x240 and the template has 91x70 pixels. The test was done only with two running processes, the capture and sign detection processes.

Method	320,240	640,480
Square difference	4.65ms	21.20ms
Cross Correlation	3.65ms	16.30ms
Correlation coefficient	4.15ms	18.00ms
Normalized		
Square difference	4.90ms	22.10ms
Cross Correlation	4.95ms	22.20ms
Correlation coefficient	5.05ms	22.90ms

Table 5.2: Average times for template matching processing. The average times presented are result of 1000 executions, carried out on a quarter of image.

During competition, the number of templates to use at the same time is 3, but can be 6 in some situations, as referred in the section 4.6. If the software needs to run 6 templates using the VGA mode and using the cross correlation method, the fastest one, the template matching takes around 100ms. Since, to improve reliability, a signal is only considered to be validly detected after three consecutive equal matches, the total identification time is at least 300ms. So, this high time spent in the process using large images, invalidates its use. In the case of the small images, for 6 templates, the spent time ranges from 21.9ms, for the cross correlation method, to 30.3ms, for the normalized correlation coefficient method. This time, although high, is acceptable.

There is no big difference among the several methods, in terms of computation time. So, the decision on which one to use should be based on accuracy. Exhaustive tests were performed to get the accuracy of each method in 4 scenarios. On each scenario, the *STOP*, *LEFT* and *FRONT* signs were tested. These are the most important ones, because its detection occurs during the run. The four scenarios correspond to real situations during a run. They are: when the car comes in front, near the crosswalk or around one meter of it, or coming from the left side and also near the crosswalk or around one meter of it.

A total of 1000 tests were done, for each position, for each method, and for each sign. The results are depicted in the table 5.3. The tests were done in a laboratory, which is different from the real arena. However, in order to create more real conditions, during the tests the lights were switched on and off and the blinds were opened and closed.

Scenario	Method	STOP	LEFT	FRONT
Front Close to the crosswalk	Square Differences	93.0%	0.0%	0.0%
	Square Differences, Normalized	95.9%	0.0%	0.0%
	Cross Correlation	0.0%	0.0%	0.0%
	Cross Correlation, Normalized	98.5%	0.0%	0.0%
	Correlation Coefficients	52.0%	99.0%	64.9%
	Correlation Coefficients, Normalized	97.5%	100.0%	90.0%
Front, one meter away of the crosswalk	Square Differences	0.0%	0.0%	0.0%
	Square Differences, Normalized	0.0%	0.0%	0.0%
	Cross Correlation	67.0%	0.0%	0.0%
	Cross Correlation, Normalized	0.0%	0.0%	0.0%
	Correlation Coefficients	47.8%	90.6%	97.7%
	Correlation Coefficients, Normalized	87.9%	93.4%	99.8%
Left, close to the crosswalk	Square Differences	57.0%	0.0%	0.0%
	Square Differences, Normalized	80.0%	0.0%	15.0%
	Cross Correlation	30.0%	0.0%	0.0%
	Cross Correlation, Normalized	97.3%	0.0%	0.0%
	Correlation Coefficients	55.9%	71.0%	65.6%
	Correlation Coefficients, Normalized	85.4%	95.0%	86.3%
Left, one meter away of the crosswalk	Square Differences	0.0%	0.0%	0.0%
	Square Differences, Normalized	0.0%	0.0%	0.0%
	Cross Correlation	0.0%	0.0%	0.0%
	Cross Correlation, Normalized	0.0%	0.0%	0.0%
	Correlation Coefficients	52.6%	44.9%	80.1%
	Correlation Coefficients, Normalized	86.7%	63.8%	89.9%

Table 5.3: Signals detection accuracy, calculated based on 1000 runs

From the results it is clear that the square differences and the cross correlation methods

(normalized or not) are useless. No matter the chosen scenario at least one of the sign was never detected. The correlation coefficient methods reveal quite acceptable results, being the normalized more accurate than the other one. The difference in accuracy between the non-normalized and the normalized methods is due to the high variations of the light, forced during the tests. This situation was expected, as referred in the section 4.6. When the method is normalized, the interference due to the lightning is reduced.

As expected the accuracy is worse when the vehicle come from the curve or when is away of the crosswalk. However, the results, at least for the normalized method, validates its usage.

5.4 Portuguese Robotics Open

The ROTA project's vehicle participated in the autonomous driving competition of the Portuguese Robotics Open, which took place in Batalha, in 2010. The high level control software that ran in *zinguer*, the ROTA's vehicle, mostly followed the global architecture described in chapter 3. The only difference resided in the vision and perceptions systems, which, by that time, March 2010, were splited in only two processes. One of the processes was responsible for the road capture, visualization and processing. All the perception elements related to the road perception were incorporated in it. The other process was responsible for the sign capture, visualization and processing.

Not all the competition's challenges were covered by the ROTA's high level software. The road maintenance area was not covered and the obstacle avoidance was only partially covered. However, disregarding these limitations the overall performance was quite good. The vehicle finished its participation in the third place. It kept all the time in lane and had a 100% success in the traffic sign identification.

Chapter 6

Conclusions

6.1 Work Summary

One of the main goals of this work was the adoption of the CAMBADA's software architecture in the ROTA's vehicle. This goal was accomplished and even overcome, since some improvements were added. First of all, the CAMBADA architecture was studied as well as the existent ROTA's software and the overall architecture was defined. Two central modules of the architecture, the RTDB (Real-Time Data Base) and the PMan (Process Manager), were reused with none or minor modifications. The RTDB required no modifications. Only a configuration file, defining the items required for the autonomous driving purposes, had to be written. The PMan module was modified to improve its performance. A configuration file was defined and used to establish the precedence relations between the different processes.

Seven different type of processes were defined: image capturing, image visualization and recording, road image processing, sign image processing, perception (Road analysis), agent (decision and action), and low-level communication handler. The image capturing process gets images from a camera or a movie file, and stores this images, in a shared memory. The image visualization and recording process can be used to display images in the screen and/or to save it into a movie file. The road image processing process extracts features from the road image and stores them into the RTDB. These features are mainly points of interest that are used to identify the delimiting lines, the crosswalk, and the obstacles. The sign image identification process uses template matching to identify signs in the traffic panels. The perception process updates the world representation based on sensor data and on the previous state of the world. The agent process is the intelligent part of the software and decides what actions to take based on the world state and on the goals. The actuation orders are stored in the RTDB. Finally, the low-level communication handler makes the connection between the high level software (the one being described) and the low-level subsystem.

The CAMBADA software architecture was developed for a cooperative team of robots. In that sense, a communication mechanism to keep the RTDB in different computer nodes synchronized exists. This raised the idea of introducing a pit box in the system, which could be used as an aiding tool during the development phase. For that, a new process appears which is responsible for the communication among computer nodes.

In what ROTA concerns, a lot o valid code has been developed in the last years and hence it was used to implement the different processes. However, in some cases a new approach was followed. For instance, the road image processing is based on ROIs, expressed in real (track)

coordinates, to define the areas of the image to be scanned.

The ROTA's vehicle participated in the autonomous driving competition of the Portuguese Robotics Open, held in Leiria, in 2010, reaching the third place. By that time, the architecture was not completely implemented, as the capturing, visualization and processing of images were incorporated in the same process, one for each camera.

6.2 Conclusions

A main conclusion of this work is that the CAMBADA architecture proved to be appropriate and abstract enough to be easily adapted to other robotic scenarios, where the goals are completely different. The CAMBADA robots play soccer while the ROTA vehicle does autonomous driving. Also, one can conclude that the different processes play similar roles in the two architectures.

Whenever was possible, configuration files were used to bring flexibility to the software. A main advantage of this is the possibility to change the execution conditions without requiring a recompilation. Configuration files were introduced in the PMan module and on the capturing process.

The vision system architecture proposed and implemented during this work proved to be a good approach. The development of any new vision application can focus only in the processing part, since the capturing and visualization are already done. This was applied in the development of the template matching test application, described in the previous chapter. Furthermore, the CAMBADA team is planning to use it in their software.

6.3 Future Directions

Several directions can be pointed out for future work in the ROTA's vehicle, both in software and hardware. In hardware, a battery monitoring module is a quite useful desired improvement. If after a change in the software the vehicle does not function properly, the cause in assigned to the change. However, sometimes, it is not the cause, since, coincidentally, the batteries may have become discharged at the same time.

In terms of the high level software there are several things left to do. The software to detect and avoid obstacles is still far from completed. An approach to identify the roadwork cones exists, but the problem of driving in that zone was not covered yet. At the perception level, the code to improve the vehicle self-localization has to be done. An approach to use a kalman filter for that purpose was initiated but not concluded. Without a proper self-localization the trajectory planner module, already implemented, cannot be properly used.

A pit-box Graphical user interface (GUI) application could be an important piece of software for the ROTA project. During development it may be used to display information about the vehicle's perception and reasoning, making the debugging phase less painful. It can also integrate control issues, like a control panel to set the traffic sign and a remote control mechanism to drive the vehicle.

Test using the real vehicle is sometimes impossible. The use of a simulation framework is often a valid alternative. This framework should simulate a track and associated elements, the vehicle's physics and the vision cameras. This way, almost all the software modules can be tested in the absence of the real vehicle. Another important advantage is given by a simulation framework. Tests in the laboratory are done in a track that only represents a

small part of the one used in competition. In the simulator the full competition track can be used.

Bibliography

[1] S.J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice hall, 2009.

[2] G.A. Bekey. *Robotics: state of the art and future challenges*. Imperial College Press, 2008.

[3] T. Bräaunl. *Embedded robotics: mobile robot design and applications with embedded systems*. Springer-Verlag New York Inc, 2008.

[4] M. Brady. Artificial intelligence and robotics. *Artificial Intelligence*, 26(1):79–121, 1985.

[5] R.N. Jazar. *Theory of applied robotics: kinematics, dynamics, and control*. Springer, 2010.

[6] L. Wang, K.C. Tan, and C.M. Chew. *Evolutionary robotics: from algorithms to implementations*. World Scientific Pub Co Inc, 2006.

[7] B. Siciliano and O. Khatib. *Springer handbook of robotics*. Springer-Verlag New York Inc, 2008.

[8] G.A. Bekey. *Autonomous robots: from biological inspiration to implementation and control*. The MIT Press, 2005.

[9] R.C. Arkin. *Behavior-based robotics*. The MIT Press, 1998.

[10] U. Nehmzow. *Mobile robotics: a practical introduction*. Springer Verlag, 2003.

[11] Robótica 2010. http://robotica2010.ipleiria.pt, November 2010.

[12] Robocup. http://www.robocup.org/, November 2010.

[13] Robótica 2010. http://www.spr.ua.pt/fnr/, November 2010.

[14] *Rules and Technical Specifications for Autonomous Driving Competition, Portuguese Robotic Open*, January 2010.

[15] José Luís Azevedo, Artur Pereira, Bernardo Cunha, and Luís Almeida. ROTA: a Robot for the Autonomous Driving Competition. In *7th Conference on Mobile Robots and Competitions – Robotica2007*, Paderne, Portugal, April 2007.

[16] Bruno Alexandre de Oliveira Pires. Projecto rota 2006, robot triciclo para condução autónoma. Master's thesis, University of Aveiro, 2006.

[17] T.L. Dean and M.P. Wellman. *Planning and control*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1991.

[18] B. Hayes-Roth. An architecture for adaptive intelligent systems. *Artificial Intelligence*, 72(1-2):329–365, 1995.

[19] M.J. Mataric. Behavior-based control: Main properties and implications. In *Proceedings, IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems*, pages 46–54. Citeseer, 1992.

[20] D. Garlan and M. Shaw. An introduction to software architecture. *Advances in software engineering and knowledge engineering*, 1:1–40, 1993.

[21] R. Brooks. A robust layered control system for a mobile robot. *IEEE journal of robotics and automation*, 2(1):14–23, 1986.

[22] V.G. Junior, S.P. Parikh, and J.O. Junior. Hybrid deliberative/reactive architecture for human-robot interaction. *Procedings of COBEM*, 2005.

[23] R.A. Brooks. Intelligence without representation. *Artificial intelligence*, 47(1-3):139–159, 1991.

[24] R.A. Brooks. Elephants don't play chess. *Robotics and autonomous systems*, 6(1-2):3–15, 1990.

[25] G. Saridis. Intelligent robotic control. *IEEE Transactions on Automatic Control*, 28(5):547–557, 1983.

[26] Maja J. Matarić. Reinforcement learning in the multi-robot domain. *Auton. Robots*, 4(1):73–83, 1997.

[27] R. Murphy. *Introduction to AI robotics*. The MIT Press, 2000.

[28] L. De Silva and H. Ekanayake. Behavior-based Robotics And The Reactive Paradigm A Survey. In *Computer and Information Technology, 2008. ICCIT 2008. 11th International Conference on*, pages 36–43. IEEE, 2009.

[29] N.J. Nilsson. A mobius automaton: an application of artificial intelligence techniques. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 509–520. Citeseer, 1969.

[30] N.J. Nilsson and SRI INTERNATIONAL MENLO PARK CA. Shakey the robot. 1984.

[31] J.H. Connell. SSS: A hybrid architecture applied to robot navigation. In *IEEE Conference on Robotics and Automation*, pages 2719–2724. Citeseer, 1992.

[32] R.A. Brooks. How to build complete creatures rather than isolated cognitive simulators. In *Architectures for Intelligence: The Twenty-second Carnegie Mellon Symposium on Cognition*, pages 225–239, 1991.

[33] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the National Conference on Artificial Intelligence*, pages 809–809. Citeseer, 1992.

[34] D. Chapman. Planning for conjunctive goals* 1. *Artificial intelligence*, 32(3):333–377, 1987.

[35] P.E. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, volume 278, pages 268–272. Seattle, 1987.

[36] R.C. Arkin. *Towards cosmopolitan robots: Intelligent navigation in extended man-made environments*. PhD thesis, University of Massachusetts, 1987.

[37] R.C. Arkin and T. Balch. AuRA: Principles and practice in review. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2):175–189, 1997.

[38] R.J. Firby. Adaptive execution in complex dynamic worlds. *Yale University, New Haven, CT*, 1989.

[39] R.P. Bonasso, R.J. Firby, E. Gat, D. Kortenkamp, D.P. Miller, and M.G. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2):237–256, 1997.

[40] N.J. Nilsson. *Artificial intelligence: a new synthesis*. Morgan Kaufmann, 1998.

[41] N.J. Nilsson. Teleo-reactive programs and the triple-tower architecture. *Electron. Trans. Artif. Intell.*, 5:99–110, 2001.

[42] J.S. Albus. Outline for a theory of intelligence. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):473–509, 1991.

[43] J.S. Albus. A Reference Model Architecture for Intelligent Systems Design., 1994.

[44] J.L. Azevedo, B. Cunha, and L. Almeida. Hierarchical distributed architectures for autonomous mobile robots: a case study. In *Proc. ETFA2007-12th IEEE Conference on Emerging Technologies and Factory Automation, Patras, Greece*, pages 973–980, 2007.

[45] L. Almeida, F. Santos, T. Facchinetti, P. Pedreiras, V. Silva, and L.S. Lopes. Coordinating distributed autonomous agents with a real-time database: The CAMBADA project. *Computer and Information Sciences-ISCIS 2004*, pages 876–886, 2004.

[46] José Ricardo Marques de Oliveira. World representation for an autonomous driving robot. Master's thesis, University of Aveiro, 2009.

[47] Arturo Bajuelos Castillo. Planeamento de trajectória em condução autónoma. Technical report, IEETA, Universidade de Aveiro, Portugal, 2009.

[48] P. Pedreiras and L. Almeida. Task management for soft real-time applications based on general purpose operating systems. *Robotic Soccer*, pages 598–607, 2007.

[49] M.S. Joshi. *C++ design patterns and derivatives pricing*. Cambridge Univ Pr, 2008.

[50] G.R. Bradski and A. Kaehler. *Learning opencv*. O'Reilly, 2008.

[51] André Prata. Automatização do cálculo do mapa de distâncias num veículo de condução autónoma. Technical report, IEETA, Universidade de Aveiro, Portugal, 2010.

[52] Miguel da Rosa Carvalhal Sequeira. Perception and intelligent localization for autonomous driving. Master's thesis, University of Aveiro, 2009.

Appendix A

Configuration Files

The configuration files has a pre-defined tree:

```
config
└── <car name>
    ├── <car name>.cfg
    ├── capture
    │   ├── <camera>.cfg ..................................................Camera configurations
    │   ├── <PMAN>.cfg ....................................................Pman Configurations
    │   └── <SHM>.cfg .....................................................Shared Images Configuration
    ├── CarBase
    │   ├── <CarBase>.cfg .................................................Car platform properties
    │   └── <steer>.map ...................................................Steering servo map
    ├── rtdb
    │   ├── <rtdb>.cfg ....................................................Database itens configuration
    │   └── <rtdb>.ini ....................................................Automatically created
    ├── signperception
    │   ├── <traffic-signs>.cfg ...........................Templates list for Traffic Lights signs
    │   ├── <vertical-signs>.cfg ..........................Templates list for Vertical signs
    │   ├── traffic-lights ................................Folder with signals templates
    │   └── vertical-signs ................................Folder with signals templates
    ├── undistort
    │   └── <Calibration Map>.xml .........................Image to World coordinates map
    └── world
        └── <map name>.cfg ................................................Track description
```

To better organize all the configuration files, a global one is defined. This one have information about each process, and lists the configuration files to be used by the modules to each process. For instance, the capture uses a configuration file, but there is two capture processes (two instances of the capture module), one to the road and other to the signs. So, may be neccessary to give diferent configurations to the cameras.

```
##################################
#       Road Camera config       #
##################################
Road :
{
  # Process Name
  ProcessName = "RoadCamera";
  # capture input:
  # - path to a movie
  # - number of camera
  Capture = 1;
  # Camera configurations
  CameraConfig = "capture/cameraDown.cfg";
  # Shared memory image configuration
  SHMConfig = "capture/roadSHM.cfg";
};

##################################
#      Signs Camera config       #
##################################
Signs :
{
  # Process Name
  ProcessName = "SignsCamera";
  # capture input:
  # - path to a movie
  # - number of camera
  Capture = 0;
  # Camera configurations
  CameraConfig = "capture/cameraUp.cfg";
  # Shared memory image configuration
  SHMConfig = "capture/signsSHM.cfg";
};

##################################
#        CarBase config          #
##################################
CarBase :
{
  ProcessName = "CarBase";
  Config = "carbase/CarBase.cfg";
};

##################################
#          World config          #
##################################
World :
{
  Config = "world/World.cfg";
};

##################################
#       Undistort config         #
##################################
Undistort :
{
  Config = "undistort/Undistort.cfg";
};
```

A.1 RTDB Configuration

```
AGENTS = carBox, zinguer;

ITEM RT_CAR_VELOCITY { datatype = RtCarVelocity; headerfile = rtdb_carbase.hpp; }

ITEM RT_CAR_LIGHTS { datatype = RtCarLights; headerfile = rtdb_carbase.hpp; }

ITEM RT_CAR_KINEMATICDATA { datatype = RtCarKinematicData; headerfile = rtdb_carbase.hpp; }

ITEM RT_CAR_SENSORSDATA { datatype = RtCarSensorsData; headerfile = rtdb_carbase.hpp; }


ITEM RT_ROAD_SEARCHCRITERIA { datatype = RtRoadSearchCriteria; headerfile = rtdb_roadvision.hpp; }

ITEM RT_ROAD_LANEDATA { datatype = RtRoadLaneData; headerfile = rtdb_roadvision.hpp; }

ITEM RT_ROAD_CROSSWALKDATA { datatype = RtRoadCrosswalkData; headerfile = rtdb_roadvision.hpp; }

ITEM RT_ROAD_OBSTACLESDATA { datatype = RtRoadObstaclesData; headerfile = rtdb_roadvision.hpp; }

ITEM RT_ROAD_RMADATA { datatype = RtRoadRMAData; headerfile = rtdb_roadvision.hpp; }

ITEM RT_ROAD_VSIGNBASEDATA { datatype = RtRoadVSignBaseData; headerfile = rtdb_roadvision.hpp; }


ITEM RT_SIGN_SEARCHCRITERIA { datatype = RtSignSeachCriteria; headerfile = rtdb_signvision.hpp; }

ITEM RT_SIGN_DATA { datatype = RtSignData; headerfile = rtdb_signvision.hpp; }

ITEM LAPTOP_INFO { datatype = LaptopInfo; headerfile = rtdb_systeminfo.hpp; }


SCHEMA driver
{
    shared = RT_CAR_VELOCITY, RT_CAR_LIGHTS, RT_CAR_KINEMATICDATA, RT_CAR_SENSORSDATA, LAPTOP_INFO
    local = RT_ROAD_SEARCHCRITERIA, RT_ROAD_LANEDATA, RT_ROAD_CROSSWALKDATA, RT_ROAD_OBSTACLESDATA,
            RT_ROAD_RMADATA, RT_ROAD_VSIGNBASEDATA,
            RT_SIGN_SEARCHCRITERIA, RT_SIGN_DATA ;
}

SCHEMA coach
{
    shared = RT_CAR_VELOCITY, LAPTOP_INFO
}

ASSIGNMENT { schema = driver; agents = zinguer; }

ASSIGNMENT { schema = coach; agents = carBox; }
```

A.2 World Map

```
Sections = (
 {
    # Section Type:
    # * "CROSSWALK"
    # * "STRAIGHT"
    # * "CURVE"
    # * "CROSSWALK"
    Type = <string>;

    % Section Name
    Name = <string>;

    # Section position and heading
    # Milimeters and rads used
    Coordinates :
    {
      X = <integer>;
      Y = <integer>;
      theta = <float>;
    };

    # Section Dimentions
    # Milimiters and/or rads
    Dimensions :
    {
      Witdh = <integer>;
      Length = <float>;
    };

    # If global existence is disable it is ignored
    ObstaclesPossible = true;

    # Obstacles list
    Obstacles = ( );

    # Lines Description
    # percentage values
    LeftLine :
    {
      start = <integer>;
      end = <integer>;
    };
    CenterLine :
    {
      start = <integer>;
      end = <integer>;
    };
    RightLine :
    {
      start = <integer>;
      end = <integer>;
    };
  }
);

# Section connection
# Assume the section description order
Connections = (
  {
    Source = 0;
    Target = 1;
    Label = "<1,0>";
  },
);
```

A.3 Process Manager

```
# List the process precedences for PMAN
# The names used are the same defined on the
# global configuration file on root config
# folder
Precedences =
(
  ( "Process A", "Process B" ),
  ( "Process B", "Process C" ),
  ( "Process X", "Process Y" )
);
```

A.4 Images Database Configurations

```
#
# Shared memory image configuration
#

# Key for shared memory
Key = 0;

# size reference
ReferenceSize :
{
  Width = 320;
  Height = 240;
};

# List of images in shared shared memory
Images :
(
  {
     # Name of the image, that also defines the format
     Name = "RAW";

     # Image scale relative to capture image
     # Is a real number that is multiplied by the ReferenceSize
     # to produce the image size.
     Scale = 1;

     # Number of channels
     \# Support: 1, 2, 3, 4
     Channels = 1;

     # Pixel Depth (in bits)
     # only implemented the unsigned forms!
     # Support: 8, 16, 32, 64
     Depth = 8;

     # Is necessary a slot with this tag with true value.
     # If none two or more has this a exception is given!
     Original = true;

     # Mark this image as automatic convertible.
     Convertible = true;
    }
);
```

A.5 Camera Configurations

```
#
# Camera configuration file
#

#
# Supported

#Support for Format 7
format7 = false;

connection =
{
  # Supported FPS
  # - 1 (1.875)
  # - 3 (3.75)
  # - 7 (7.5)
  # - 15
  # - 30
  # - 60
  # - 120
  # - 240
  fps = 30;

  # Supported spped
  # - 100
  # - 200
  # - 300
  # - 800
  # - 1600
  # - 3200
  speed = 400;

  # Supported Image Formats
  # *640x480
  #  -YUV411 66
  # -YUV422 67
  # -RGB 68
  # -MONO 69
  # *320x240
  # -YUV422 65
  # * RAW8
  #  - Format 7 361
  format= 69;

  # Support modes
  # 0 - Legacy
  # 1 - 1394B
  mode = 0;
};

# Used only with format 7
CCD =
{
  nCols = 640;
  nRows = 480;
  ccdCol = 0;
  ccdRow = 0;
  centerCol = 320;
  centerRow = 240;
  inRadius = 70;
  outRadius = 200;
  nColors = 9;
};
```

```
feature =
{
  brightness = 209;
  exposure = 322;
  gain = 124;
  gamma = 0;
  saturation = 99;
  sharpness = 92;
  shutter = 3;
  white_balance_red = 86;
  white_balance_blue = 172;
  autoBrightness = false;
  autoExposure = false;
  autoGain = false;
  autoSaturation = false;
  autoShutter = false;
  autoWhiteBalance = false;
};
```

A.6 Signs templates list

```
signal =
(
  {
    # complete path to the image
    path = <string>;

    # Template match mathed to this template
    # Available methods:
# * SQDIFF
# * SQDIFF_NORMED
# * CCORR
# * CCORR_NORMED
# * CCOEFF
# * CCOEFF_NORMED
    method = <string>;

    # The area where this can be searchable
    pos =
    {
      x = <integer>;
      y = <integer>;
      witdh = <integer>;
      height = <integer>;
    };
  }
);
```